O'REILLY®

TURING 图灵程序设计丛书

Presto 实战

Presto: The Definitive Guide



马特·富勒
[美] 曼弗雷德·莫泽 著
马丁·特拉韦尔索
张晨 黄鹏程 傅宇 译


中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

译者简介

张晨

毕业于上海交通大学，热爱大数据技术、数据库、分布式系统和函数式编程，现于Indeed东京担任软件工程师。

黄鹏程

毕业于北京邮电大学，专注于数据库和大数据平台的开发与架构工作。曾就职于中国民生银行，担任大数据基础架构团队负责人。现任阿里云高级产品专家，负责阿里云数据库相关产品的设计与规划工作。

傅宇

毕业于南京大学，专注于数据库技术，现任阿里云技术专家，担任PolarDB-X云原生分布式数据库内核研发工作，熟悉分布式事务、查询优化器和执行器，对大数据领域充满热情。

数字版权声明


图灵程序设计丛书

Presto实战

Presto: The Definitive Guide

[美] 马特·富勒　曼弗雷德·莫泽　马丁·特拉韦尔索 著

张晨　黄鹏程　傅宇 译

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社

北　京

图书在版编目（CIP）数据

Presto实战 / （美）马特·富勒（Matt Fuller），（美）曼弗雷德·莫泽（Manfred Moser），（美）马丁·特拉韦尔索（Martin Traverso）著；张晨，黄鹏程，傅宇译. -- 北京：人民邮电出版社，2021.3
（图灵程序设计丛书）
ISBN 978-7-115-56005-6

Ⅰ. ①P… Ⅱ. ①马… ②曼… ③马… ④张… ⑤黄… ⑥傅… Ⅲ. ①数据处理 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2021)第029514号

内 容 提 要

随着各行各业对大数据实时查询的需求持续增长，数据查询及分析引擎正变得不可或缺。Presto 是由 Facebook 开源的高性能分布式 SQL 查询引擎，其用户包括 Netflix、Airbnb、LinkedIn、Twitter、Uber 等知名公司。本书由 Presto 的核心开发人员参与撰写，教你系统地学习 Presto 的用法。书中内容涵盖 Presto 的安装、设计理念、查询操作、最佳实践、与主要云平台的结合等。本书分为三大部分：第一部分介绍 Presto 的基础知识；第二部分更进一步，介绍 Presto 架构、集群部署、与数据源的连接等；第三部分讲解安全配置以及 Presto 的实际用例。你可以通过本书学会针对不同的数据源快速执行交互式 SQL 数据分析，并利用 Presto 管理和使用海量数据。

本书适合数据分析师和其他大数据从业人员阅读。

◆ 著　　[美] 马特·富勒　曼弗雷德·莫泽
　　　　马丁·特拉韦尔索
译　　张　晨　黄鹏程　傅　宇
责任编辑　温　雪
责任印制　周昇亮
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　https://www.ptpress.com.cn
北京　　　　印刷
◆ 开本：800×1000　1/16
印张：16.5
字数：390千字　　2021年3月第1版
印数：1－2 500册　　2021年3月北京第1次印刷
著作权合同登记号　图字：01-2020-4163号

定价：99.00元
读者服务热线：(010)84084456　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东市监广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O’Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O’Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

本书赞誉

“这本书介绍了何谓 Presto，以及能让你将其运用自如的所有关键知识。”

——Dain Sundstrom 和 David Phillips
Presto 之父，Presto 软件基金会发起人

“Presto 在 Pinterest 的数据分析中发挥了关键作用，你可以通过这本书学习从使用场景到如何大规模运行 Presto 集群在内的重要知识。”

——Ashish Kumar Singh
Pinterest 大数据查询处理平台技术负责人

“对于现代云架构，无论是社区构建还是数据的快速分析处理技术，Presto 都设置了很高的标杆。如果想构建现代化的分析技术栈，那么这本书值得一读。”

——Jay Kreps
Apache Kafka 联合创作者，Confluent 联合创始人和首席执行官

“Presto 免除了编写代码来管理分布式查询处理的负担，从而为学术界和工业界的从业者节省了大量时间。从如此高质量的开源分布式 SQL 引擎入手，我们可以专注于在新领域中的创新工作，而不必为每一个新的分布式数据系统项目重新发明轮子，对此我们十分感激。”

——Daniel Abadi
马里兰大学帕克分校计算机科学系教授

“近些年，Presto 社区成长迅速。作为又一款 SQL-on-Hadoop 查询引擎，它兼具优秀的性能、易用的接口和简洁的设计。国内外包括阿里巴巴在内的许多公司使用它，其优良的架构也让用户很容易进行定制和扩展。这本书既可以作为学习 Presto 的实战入门指南，也可以当作手册供随时查阅。无论你之前是否使用过 Presto，相信你都能从中受益。”

——曹伟

PolarDB 创始人，阿里巴巴研究员

“Presto 如何超越一时轰动的 Impala 和拥趸众多的 Spark 成为交互式分析的龙头？带着这个疑问，我快速看完了全书，颇有点后知后觉、相见恨晚的感觉。Presto 一改大数据的窠臼，基于 SQL-on-Anything 的理念和开源开放的工程实践对接大小各种数据源，降低了解决实际问题的门槛，难怪大家都喜欢。这本书翻译流畅，紧跟业界进展。开源大数据爱好者可以先不要急着掉进源代码细节里，而是从这本书的内容起步，从问题和场景入手，搞懂大数据。”

——郑锴

Apache Hadoop PMC 成员，阿里巴巴高级技术专家

“Presto 引擎在大数据领域的重要性不言而喻，但参考资料屈指可数，这本书正是大家期待的那本‘官方指南’。无论是 SQL 编写、技术调研、运维部署，还是二次开发，都值得一读。书中第三部分为企业级应用做了详细解答，是一大亮点。”

——腾讯 Presto Oteam 团队

目录

第二部分 深入理解 Presto

第三部分 Presto 的实际应用

译者序

对大数据领域感兴趣的读者恐怕已经对它的历史非常熟悉了：最初，由 Google 发表的关于 MapReduce、GFS 和 BigTable 等主题的论文衍生出了由 Hadoop、HDFS、HBase 等组件组成的 Hadoop 生态系统。之后，包括 Apache Hive、Apache Spark 和 Apache Flink 在内的各种工具如雨后春笋一般涌现出来。它们有的适合用来进行批数据处理，有的可以进行实时数据分析，有的支持分布式事务，满足了大数据处理方方面面的需求。在这些系统中，有一类支持 SQL 数据查询的数据处理系统非常重要：这类系统允许用户使用标准 SQL 对其数据仓库中的数据进行查询，无论是技术人员还是数据分析师都可以方便地使用。

Presto 正是这一类系统中的佼佼者。

说到 Hadoop 生态系统中的 SQL 查询处理引擎，就不得不先提到 Apache Hive —— 一个可以将用户输入的 SQL 查询翻译成如 MapReduce、Spark 等计算模型的任务，并在集群中调度和执行的系统。长期以来，Apache Hive 都是在 Hadoop 集群上执行 SQL 查询的标准组件。

本书所介绍的 Presto 和 Apache Hive 一样，都是从 Facebook 内部孵化出来的。它早期的设计目标就是解决 Apache Hive 的一些主要问题，尤其是要提高查询的执行性能。正因为如此，Presto 在很多方面兼容 Apache Hive，但又有所不同。在本书中，你可以详细了解到二者设计上的异同和产生的效果。简单来说，Presto 提供了一个查询性能远超 Apache Hive，适用于交互式查询和快速数据处理的平台。它还支持联邦查询，使得用户可以将分散在系统各处的数据孤岛整合起来，成为统一的数据查询接入点，方便用户管理各类数据。在很多企业和组织里，Presto 已经取代 Apache Hive 成为各个部门接入数据系统的统一接口，每天满足成千上万次的查询需求。

本书的第一部分提供了对 Presto 的简要介绍和使用说明，你可以从中了解到 Presto 的历史发展、主要功能以及如何简单地安装和使用它。本书的第二部分是为进阶用户准备的，它介绍了 Presto 的内部构造、基本原理、部署配置、连接器和 SQL 语句的使用等。这部分内容全面而详细地介绍了使用 Presto 的各种细节，是本书的精华所在。学习完这一部分的知

识，你应该具备正确使用和调优 Presto 的大部分技能。这一部分也很适合作为速查手册使用。本书的第三部分提供了在生产环境部署 Presto 必备的知识，帮助你配置安全特性、与其他工具集成和进行集群部署，还提供了一些真实世界中的部署案例。建议即将在生产环境中部署 Presto 的读者将这一部分作为参考。

在阅读完本书后，你不但可以编写适合于 Presto 执行的 SQL 查询，熟悉 Presto 运维方面的各种知识，而且还可以对 Presto 的内部构造有所了解。希望本书能为对 Presto 感兴趣的读者提供一份好的实战指南，帮助各位读者理解 Presto 的方方面面。

Presto 的繁荣发展离不开它的开源社区：在 Facebook 内部上线之后不久它就宣告开源。实际上，Presto 从开发伊始就将开源作为目标之一。现在，Presto 丰富的生态系统已经被构建出来，这个生态系统包括数不清的开源插件和库、提供支持服务的组织和企业，以及云服务提供商等。目前，Presto 的开源版本主要有两个分支，一个是由 Facebook 主导的 PrestoDB 项目，另一个是由 Presto 软件基金会和部分 Presto 创始人维护的 PrestoSQL 项目。值得注意的是，后者已于 2020 年 12 月更名为 Trino 并继续发展。本书的主要作者来自 PrestoSQL/Trino 项目，所讨论的主要内容也基于 PrestoSQL/Trino 项目的实现。目前来看，这两个项目还并未产生大的分化，本书中大部分的讨论可以同时应用在这两个项目上，但仍建议你在阅读此书和开始使用 Presto 时仔细分辨项目和版本的差别。

最后，还要特别感谢在本书翻译和出版过程中参与审校、排版和付出其他辛勤劳动的各位工作人员。是你们的努力使得本书可以以如此完整的面貌呈现给读者。

张晨　黄鹏程　傅宇

2021 年 1 月

序

回顾往昔，这是多么令人惊讶的旅程！ 2012 年在 Facebook 启动 Presto 这个项目时，我们坚定地认为我们将创造一个有用的产品。我们从一开始就计划将其变成一个成功的开源项目并建立活跃的社区。2013 年，我们以 Apache 许可证开源了 Presto。

然而，从那时起，Presto 走过的路程已远超我们的想象。我们为这个项目的社区所取得的成就感到骄傲，但更重要的是，社区所获得的积极回应和帮助令我们十分感动。

Presto 的成长突飞猛进，也为其社区的用户做出了很多贡献。你可以在世界各地找到 Presto 的社区成员，Presto 的开发者则分布在巴西、加拿大、中国、德国、印度、以色列、日本、波兰、新加坡、美国、英国以及许多其他国家和地区。

在 2019 年年初启动的 Presto 软件基金会是另一个重要的里程碑。这个非营利组织致力于 Presto 这一开源分布式 SQL 引擎的发展和进步，并确保此项目在接下来的几十年保持开源、协作和独立的状态。

今天，在 Presto 软件基金会启动一年之后，回顾过去一年的发展，可以看到社区变得更加壮大，所做的贡献也越来越多。

很高兴在 O'Reilly 出版社的帮助下，马特、曼弗雷德和马丁三位作者写了这本有关 Presto 的书。书中对 Presto 做了非常精彩的介绍，并传授了开始使用 Presto 所需的所有知识。

请享受这段深入理解 Presto 及其相关领域的旅程，相关领域包括商业智能、报表、仪表盘创建、数据仓库、数据挖掘、机器学习等。

此外，别忘记探索我们在 Presto 官方网站和社区聊天室等处提供的额外资源和帮助。

Presto 社区欢迎你加入！

Dain Sundstrom 和 David Phillips
Presto 项目和 Presto 软件基金会创始人

前言

关于本书

本书是有关 Presto 分布式查询引擎的第一本也是十分重要的一本书，面向初学者和已经在使用 Presto 的用户。理想情况下，你已经对数据库和 SQL 有一定的了解，但如果没有相关知识，你也可以在学习本书的过程中查阅相关内容。无论你的专业程度如何，我们都相信你能从本书中学到一些新知识。

本书的第一部分首先介绍 Presto，之后讲解如何快速启动和运行 Presto，以便开始学习使用它，具体内容包括命令行界面的安装和初次使用，基于 JDBC 的其他客户端应用程序和 Web 应用程序（如 SQL 数据库管理或仪表盘和报表工具）的初次使用。

第二部分详细介绍 Presto 架构、集群部署、到数据源的诸多连接器，以及 Presto 的主要功能——使用 SQL 查询任何数据源信息。

第三部分介绍在生产环境中运行和部署 Presto 集群时需要了解的其他方面。具体内容包括 Web UI 的使用、安全配置，以及一些组织使用 Presto 的真实案例。

排版约定

本书使用以下排版约定。

黑体

　表示新术语。

等宽字体（`constant width`）

　表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

等宽粗体（**`constant width bold`**）

　　表示应由用户输入的命令或其他文本。

等宽斜体（*`constant width italic`*）

　　表示应由用户输入的值或根据上下文确定的值替换的文本。



此图标表示提示或建议。



此图标表示一般注记。



此图标表示警告或警示。

代码示例、授权和引用说明

本书的补充材料在 1.4.6 节有详细介绍。

如果你有技术问题，或对示例代码的使用有疑问，可以通过社区聊天室（参见 1.4.3 节）联系我们，或在本书的仓库中提交 issue（问题）。

本书是要帮你完成工作的。通常，你可以在程序和文档中直接使用本书提供的示例代码。除非你要使用大量代码，否则无须联系我们获得许可。例如，编程时用到书中的几个代码片段无须获得许可，但将示例代码出售或传播则需要获得许可；引用示例代码来回答问题无须获得许可，但将大量示例代码用于产品文档中则需要获得许可。

我们很希望但不强制要求你在引用本书的内容时加上引用说明。引用说明通常包括：标题、作者、出版社和 ISBN。例如“*Presto: The Definitive Guide* by Matt Fuller, Manfred Moser, and Martin Traverso (O’Reilly). Copyright 2020 Matt Fuller, Martin Traverso, and Simpligility Technologies Inc., 978-1-492-04427-7”。

如果你觉得自己对书中示例代码的使用超出了上述授权的范围，请通过 permissions@oreilly.com 联系我们。

O'Reilly在线学习平台（O'Reilly Online Learning）

O'REILLY®

40 多年来，O'Reilly Media 致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独一无二的专家和革新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O'Reilly 的在线学习平台允许你按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O'Reilly 和 200 多家其他出版商提供的大量文本和视频资源。有关的更多信息，请访问 https://oreilly.com。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。可以通过链接 https://oreil.ly/PrestoTDG 访问该页面。

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com。

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia。

致谢

我们要感谢 Presto 社区的每个人，感谢你们使用 Presto、推广它、帮助其他的用户、为项目做贡献，甚至提交代码和文档。作为社区的一员，我们十分激动，并期待未来大家一起获得更多的成功。

Presto 社区至关重要的一部分是 Starburst 公司。我们要感谢 Starburst 每一个人的帮助，感谢 Starburst 为 Presto 项目、使用 Presto 的客户以及本书作者（也是 Starburst 团队成员）所做的工作，以及提供的资源、支持和稳定性。

关于本书，我们要特别感谢所有在想法、反馈和审阅方面提供过帮助的人。下面是一个可能并不完整的人员列表：

Anu Sundarsan、Dain Sundstrom、David Phillips、Grzegorz Kokosiński、Jeffrey Breen、Jess Iandiorio、Justin Borgman、Kamil Bajda-Pawlikowski、Karol Sobczak、Kevin Kline、Megan Sifferlen、Neeraj Soparawala、Piotr Findeisen、Raghav Sethi、Thomas Nield、Tom Nats、Will Morrison 和 Wojciech Biela。

此外，本书的三位作者还想要表达他们个人的感激之情。

马特想感谢他的妻子 Meghan 和三个孩子 Emily、Hannah 和 Liam，感谢他们在马特写作本书时表现出的耐心和给予的鼓励。孩子们对父亲成为图书作者的激动之情帮助马特度过了许多漫长的周末和深夜。

曼弗雷德想感谢他的妻子 Yen 和三个儿子 Lukas、Nikolas 和 Tobias，感谢他们不但容忍了曼弗雷德这个技术迷，而且和曼弗雷德一样对技术、写作、学习和教学充满兴趣和热情。

马丁想感谢他的妻子 Melina 和四个孩子 Marcos、Victoria、Joaquin 和 Martina，感谢他们在过去七年里支持马丁从事 Presto 相关工作并表现出极大的热情。

电子书

扫描如下二维码，即可购买本书中文版电子版。



第一部分

Presto入门

Presto 是一个 SQL 查询引擎，它使得用 SQL 访问任何数据源成为可能。你可以使用 Presto 通过水平扩展查询处理的方式来查询大型数据集。

在第一部分中，你将了解 Presto 及其应用场景，然后学习 Presto 的安装和运行，最后了解可以连接到 Presto 并查询数据的工具。这一部分将使用一个最小化的 Presto 安装，以便你尽快开始使用它。

第1章 Presto介绍

你也许听说过 Presto[1]，然后找到了本书，抑或只想浏览第 1 章的内容，以便决定是否继续研读。本章将讨论快速增长的海量数据带来的诸多问题，以及隐藏在这些数据中的价值。Presto 是处理这些数据的关键工具，让你可以使用已证明十分成功的结构化查询语言（structured query language，SQL）及其相关工具来访问数据。

Presto 的设计和特性能让你更深入地理解数据。你不但可以更快地获得见解，而且还可以获得之前因成本太高或耗时太长而无法获得的信息。同时，你使用了更少的资源并因此节省了预算。利用节省下来的资源，你还能从数据中获得更多见解！

虽然本书也会介绍很多外部资源，但我们希望你可以先从本书开始学习。

1.1 大数据带来的问题

每个人都在采集越来越多的数据，这些数据涉及设备指标、用户行为跟踪、商业交易、地理位置、软件及系统测试过程和工作流等。通过理解和使用这些数据所获得的洞见能够决定一项计划乃至一家公司的成败。

同时，数据存储机制日益多样：关系数据库、NoSQL 数据库、文档数据库、键值存储和对象存储系统等。对于当今的组织机构，它们当中很多是必备的，只使用其中一种系统不再可行了。应对如图 1-1 所示的情形是一项令人生畏的艰巨任务。

注 1：Presto 包含两个分支：PrestoDB 和 PrestoSQL。2020 年 12 月，PrestoSQL 正式更名为 Trino。本书内容适用于所有 Presto 分支。——编者注

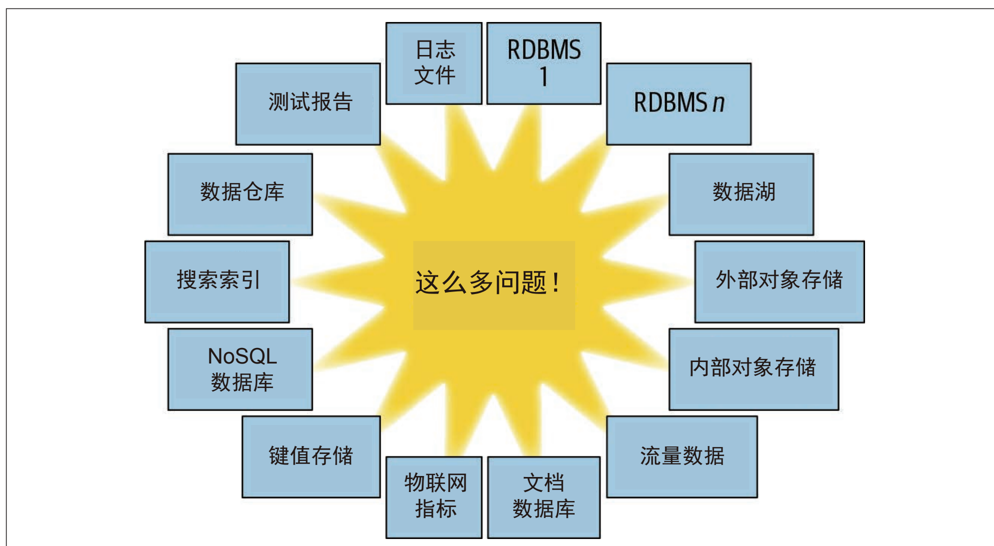


图 1-1：大数据可能会让人难以招架

此外，这些系统不允许你使用标准工具来查询和检视数据。面向特定系统的查询语言和分析工具比比皆是。与此同时，你的商业分析师已习惯使用业界标准——SQL，无数强大的工具依赖 SQL 来分析数据、创建仪表盘、制作富文本报告以及完成其他商业智能工作。

数据分散在各个孤岛上，其中有些系统甚至不能提供满足分析所需的查询性能，其他一些系统则将数据存储在单一庞大的系统上，因而不能像现代的云应用程序一样横向扩展。没有这样的能力，你就只能缩小潜在的使用场景和用户数量，因此降低了数据的实用价值。

对全世界的组织来说，创建和维护大型专用数据仓库的传统方法成本高昂。通常，对很多用户和使用模式来说，这种方法也显得缓慢且笨拙。

显而易见的是，一个系统如果可以释放所有这些价值，就将带来巨大的机会。

1.2　Presto来救场

Presto 能解决上述所有问题。通过支持不同系统上的联邦查询、并行查询和横向集群扩展等功能，它还为我们提供了更多可能性。Presto 项目的标志如图 1-2 所示。



图 1-2：Presto 的标志

Presto 是一个开源的分布式 SQL 查询引擎，它是为了高效查询不同系统和各种规模（从 GB 级到 PB 级）的数据源而从头开始设计和编写的一套系统。使用 Presto，人们不需要在快速但昂贵的商业解决方案和缓慢（且需要更多硬件）但“免费”的解决方案之间做出选择。

1.2.1 为性能和规模而生

Presto 使用分布式执行来快速查询海量数据。如果有 TB 级乃至 PB 级的数据需要查询，你可能会使用 Apache Hive 等工具，这些工具基于 Hadoop 和 Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS）工作。与这些工具相比，Presto 可以更高效地查询数据。

分析师应该使用 Presto，因为他们期望 SQL 查询可以在几毫秒（实时数据分析）、几秒或几分钟内返回结果。Presto 支持 SQL，SQL 通常用在数据仓储、数据分析、海量数据聚合和报表生成等任务上，这些任务通常被归类为**联机分析处理**（online analytical processing，OLAP）。

尽管 Presto 能理解并高效地执行 SQL，但它**并不是**一个数据库，因为它并不包含自己的数据存储系统。它无意成为一个通用关系数据库，并取代 Microsoft SQL Server、Oracle 数据库、MySQL 或 PostgreSQL。此外，Presto 也不适用于**联机事务处理**（online transaction processing，OLTP）。许多为数据仓储和数据分析任务设计和优化的数据库也是如此，如 Teradata、Netezza、Vertica 和 Amazon Redshift 等。

Presto 同时使用了众所周知的技术和新颖的技术来执行分布式查询，这些技术包括内存并行处理、跨集群节点管线执行、多线程执行模型（以充分利用所有 CPU 核心）、高效的扁平内存数据结构（以最小化 Java 的垃圾回收）和 Java 字节码生成等。本书不会详细介绍 Presto 内部的复杂实现。借助上述技术，Presto 用户可以以比其他方案更少的成本更快地获得查询结果。

1.2.2 SQL-on-Anything

Presto 的设计初衷是用来查询 HDFS 上的数据。稍后可以看到，它可以非常高效地完成这一任务，而且并未止步于此。它也可以从对象存储系统、关系数据库管理系统（RDBMS）、NoSQL 数据库和其他系统中查询数据，如图 1-3 所示。

Presto 在原地查询数据，无须事先将数据迁移集中到某个位置。因此，Presto 不仅可以查询 HDFS 和其他分布式对象存储系统中的数据，而且还可以查询 RDBMS 和其他数据源。无论数据存放在何处，Presto 都可以查询，因此它可以取代传统、昂贵和笨重的抽取 – 变换 – 加载（ETL）过程，至少可以帮你减轻相关任务的负担。显然，Presto 并非另一个 SQL-on-Hadoop 解决方案。

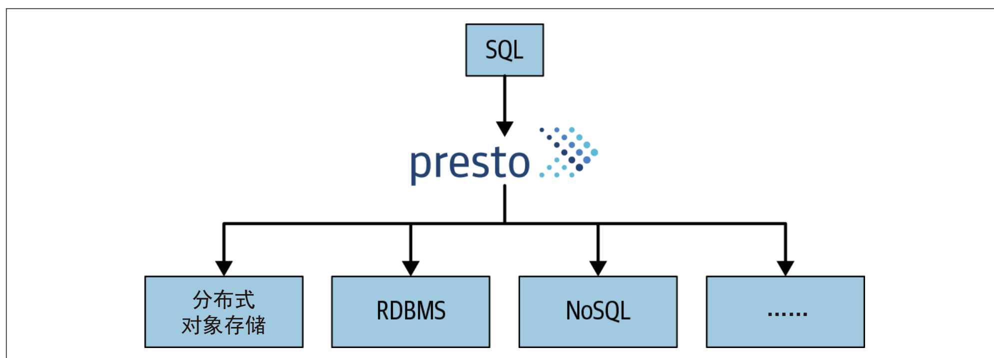


图 1-3：Presto 对多种数据源的 SQL 支持

对象存储系统包括 Amazon Web Services（AWS）提供的 Simple Storage Service（S3）、Microsoft Azure Blob Storage、Google Cloud Storage 和 S3 兼容的存储系统（如 MinIO 和 Ceph）。Presto 可以查询传统的 RDBMS，如 Microsoft SQL Server、PostgreSQL、MySQL、Oracle、Teradata 和 Amazon Redshift，还可以查询 NoSQL 数据库系统，如 Apache Cassandra、Apache Kafka、MongoDB 和 Elasticsearch。Presto 几乎可以查询任何东西，是一个真正的 SQL-on-Anything 系统。

对于用户来说，这意味着他们在与特定系统内的数据进行交互时，无须再依赖特定的查询语言和工具。他们可以简单地利用 Presto 和已有的 SQL 技能，利用已经十分熟悉的基于 SQL 的数据分析、仪表盘和报表工具，查询那些之前被锁定在分离系统中的数据。有了 Presto，用户甚至可以执行跨越多个系统的 SQL 查询。

1.2.3 存储与计算分离

Presto 没有自己的存储，它只是在数据所在之处进行查询处理。使用 Presto 时，存储和计算是分离的，它们可以各自独立地扩展。Presto 代表计算层，底层的数据源代表存储层。

这使得 Presto 可以基于对数据的分析需求来扩展和缩减计算资源。你无须移动数据或根据当前查询的需求预设计算资源和存储资源，也无须随着查询需求的变化来定期变更资源的分配。

Presto 可以通过动态扩展计算集群的规模来扩展查询能力，并可以在数据源中数据所在的位置查询数据。借助这一特性，你可以极大地优化硬件资源需求并降低成本。

1.3 Presto使用场景

Presto 灵活强大的特性使你可以自行决定如何使用它以获得最佳收益。你可以从仅用它解决一个特定的小问题开始，实际上，大部分 Presto 用户是这样开始的。

一旦你或者你所在组织中的其他 Presto 用户熟悉了这些好处和特性，你们将会发现新的应用场景。口口相传，很快你就会发现 Presto 满足了访问各种数据源的各种需求。

在下面的几节中，我们将讨论几个使用场景。记住，你可以将 Presto 应用到所有场景。此外，仅使用 Presto 解决某个特定问题也完全没有问题，只要准备好爱上 Presto 并在之后频繁使用它即可。

1.3.1 单一的SQL分析访问点

RDBMS 和 SQL 已经出现了很长时间且被证明是非常有用的。没有它们，任何组织都无法运作。实际上，很多公司会运行多个系统。Oracle Database 和 IBM DB2 这样的大型商业数据库可能支撑着你的企业软件。开源系统，如 MariaDB 和 PostgreSQL，可能用于其他解决方案或者一些内部应用程序。

作为一个消费者和分析师，你可能会遇到数不清的问题。

- 有时你甚至不知道数据在哪儿，只有凭借公司某个部门的内部知识或组织内多年的工作经验，你才能找到正确的数据。
- 为了查询多个数据库，你需要使用不同的连接和运行多种 SQL 方言的不同查询。这些查询看起来很相似，行为上却不同，因此你会感到困惑，并需要了解其中的细节。
- 若不使用数据仓库，就无法使用查询合并来自不同系统的数据。

Presto 可以帮你避免这些问题。你可以从 Presto 这里访问所有这些数据库。

可以使用一个 SQL 标准来查询所有的系统——Presto 支持的标准化 SQL、函数和运算符。

所有的仪表盘和分析工具以及其他商业智能系统，都可以指向一个系统——Presto，并访问组织当中的所有数据。

1.3.2 数据仓库和数据源系统的访问点

当一个组织需要更好地理解和分析存放在无数 RDBMS 中的数据时，就可以创建和维护数据仓库系统。从多个系统中抽取的数据通过一个复杂的 ETL 过程（通常使用长时间的批处理任务），最终进入一个严格受控的、巨大的数据仓库。

尽管数据仓库在很多情况下非常有用，但作为一个数据分析师，你会面临很多新问题。

- 除了原来的那些数据库，你的工具和查询现在又多了一个数据接入点。
- 你今天就要用的数据还没放入数据仓库。加载数据的过程痛苦、昂贵又障碍重重。

Presto 允许你添加任何数据仓库作为数据源，就像其他关系数据库一样。

如果想深入研究数据仓库的查询，可以在 Presto 里直接完成。你也可以在这里访问数据仓库及其源数据库系统，甚至可以编写将它们组合在一起的查询。Presto 让你可以使用单个系统查询任何数据库（包括数据仓库、源数据库和任何其他数据库）。

1.3.3　提供对任何内容的SQL访问

只使用 RDBMS 的日子早已一去不复返。数据现在存储在许多针对特定使用场景进行优化的不同系统中。基于对象的存储、键值存储、文档数据库、图数据库、事件流系统和其他所谓的 NoSQL 系统提供了独有的特性和优势。

至少你的组织中使用了其中的一些系统，所保存的数据对理解和改进业务至关重要。

当然，所有这些系统还要求你使用不同的工具和技术来查询和处理数据。

至少，这是一项学习曲线陡峭的复杂任务。更有可能的是，你最终只触及了数据的表层，而并没有真正获得完整的理解。你缺乏查询数据的好方法，更详细地可视化和分析数据的工具很难找或根本不存在。

另外，Presto 允许你将所有这些系统作为数据源进行连接。它使用标准的 ANSI SQL 和使用 SQL 的所有工具对外暴露要查询的数据，如图 1-4 所示。

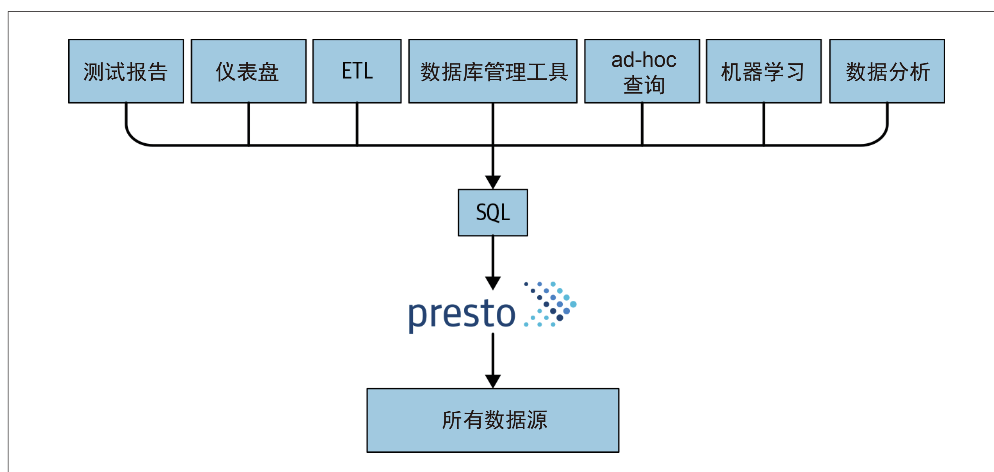


图 1-4：用于所有数据源的多个使用场景的单一 SQL 访问点

得益于 Presto，理解所有这些截然不同的系统中的数据变得更加简单，甚至第一次成为可能。

1.3.4 联邦查询

将所有的数据孤岛都暴露给 Presto 是向理解数据迈出的一大步。你可以使用 SQL 和标准工具来查询所有这些内容，但你想要回答的问题往往需要你潜入这些数据孤岛，将它们的一些侧面抽取出来，然后在同一个地方将其组合起来。

可以通过 Presto 的联邦查询满足上述需求。**联邦查询**是在一个语句中引用并使用不同数据库和模式（schema）的 SQL 查询，这些数据库和 schema 来自于完全不同的系统。在同一条 SQL 查询中，可以查询 Presto 中可用的所有数据源。

你可以定义对象存储中的用户追踪信息和 RDBMS 中客户数据之间的关系。如果键值存储包含更多相关信息，你也可以把它添加进查询。

使用 Presto 的联邦查询，你可以获得通过其他方式无法获得的洞见。

1.3.5 虚拟数据仓库的语义层

数据仓库系统为用户创造了巨大的价值，对组织来说却是一个负担。

- 运行和维护数据仓库是一个巨大且昂贵的项目。
- 需要专门的团队运行与管理数据仓库和相关的 ETL 过程。
- 将数据导入数据仓库需要用户执行烦琐的操作，并且通常很耗时。

此外，Presto 可用作虚拟数据仓库。使用这一工具和标准 ANSI SQL，你就可以定义语义层。一旦所有的数据库都设置成 Presto 的数据源，你就可以查询它们。Presto 提供了查询这些数据库所需的计算能力。使用 SQL 和 Presto 支持的函数和运算符，你可以直接从数据源获得想要的数据。在使用数据进行分析之前，无须复制、移动或转换它们。

多亏了对所有连接数据源的标准 SQL 支持，你能以更简单的方式创建所需的语义层，以支持来自工具和终端用户的查询。这一语义层覆盖了所有底层的数据源，因此无须迁移任何数据。Presto 可以在数据源和存储层查询数据。

将 Presto 用作这种“动态数据仓库”为组织提供了增强现有数据仓库的可能，甚至可以完全避免构建和维护数据仓库。

1.3.6 数据湖查询引擎

数据湖（data lake）通常指的是一个巨大的 HDFS 或类似的分布式对象存储系统，在数据被转储到这些存储系统时，并没有特别考虑接下来应如何访问它们。Presto 可以使它们成为有用的数据仓库。实际上，Facebook 开发 Presto 的目的就是对一个非常大的 Hadoop 数据仓库进行更快和更强大的查询，提供 Hive 和其他工具无法提供的能力。这也是 Presto Hive 连接器的起源，6.4 节将讨论这个连接器。

现代数据湖通常使用 HDFS 之外的其他对象存储系统，这些系统来自云供应商或其他开源项目。Presto 能使用 Hive 连接器连接它们，无论数据在哪里、如何存储，你都可以在数据湖上使用基于 SQL 的数据分析。

1.3.7 SQL转换和ETL

基于对 RDBMS 和其他类似数据存储系统的支持，Presto 也可用于迁移数据。它所提供的丰富的 SQL 函数，可以让你查询数据、转换数据，并将数据写入同一个数据源或任何其他数据源。

在实践中，这意味着你可以从对象存储系统或键值存储中复制数据到 RDBMS，并进一步分析。当然，你也可以转换和聚合数据，以获得新的理解。

反过来，从运转中的 RDBMS 或 Kafka 这样的事件流系统当中读取数据，并将它们移动到数据湖中的操作也很常见，这可以减轻多个分析师对 RDBMS 进行查询的负担。ETL 过程（现在也常称为数据准备过程）可作为这一操作的重要组成部分，以提升数据质量并创建更适合查询和分析的数据模型。

在这一场景下，Presto 是整个数据管理解决方案的关键部分。

1.3.8 更快的响应带来更好的数据见解

复杂的问题和海量数据集带来了诸多限制。将数据复制并加载到你的数据仓库并在其中分析它们最终会过于昂贵。计算可能消耗太多的计算资源而无法处理全部数据，或者要耗费数天才能得到答案。

Presto 一开始就避免了数据复制。Presto 的并行计算和重度优化通常能为你的数据分析带来性能提升。

如果原来需要 3 天的查询现在只需 15 分钟就可以完成，那么执行这个查询便是有价值的。从这些结果中获得的知识让你可以执行更多的查询。

总之，Presto 更快的处理速度带来了更好的数据分析和结果。

1.3.9 大数据、机器学习和人工智能

Presto 向围绕 SQL 的标准工具暴露了越来越多的数据，并将查询扩展到海量数据集上，这使其成了大数据处理的主要工具。如今，大数据处理通常还包括统计分析，由于机器学习和人工智能系统的发展，处理的复杂度也随之上升。基于对 R 语言和其他工具的支持，Presto 也必将在这些使用场景中占据一席之地。

1.3.10 其他使用场景

在前面的章节里，我们提供了对 Presto 使用场景的概览。新的场景和组合不断涌现。

在第 13 章中，你可以学习到一些知名公司和组织使用 Presto 的细节。我们将这些信息留在本书的结尾，这样你就可以先阅读接下来的章节，以获得理解眼前的数据所必备的知识。

1.4 Presto资源

在本书之外，还有很多可用资源可以帮你扩展有关 Presto 的知识。这一节列举一些重要的起始点，其中大多数包含丰富的信息并指向更多的资源。

1.4.1 官方网站

Presto 软件基金会负责管理 Presto 项目的开源社区并维护项目网站。可以在图 1-5 中看到 Presto 官方网站的主页。Presto 网站中包含文档、联系信息、发布最新消息和事件的社区博客，以及很多其他信息。

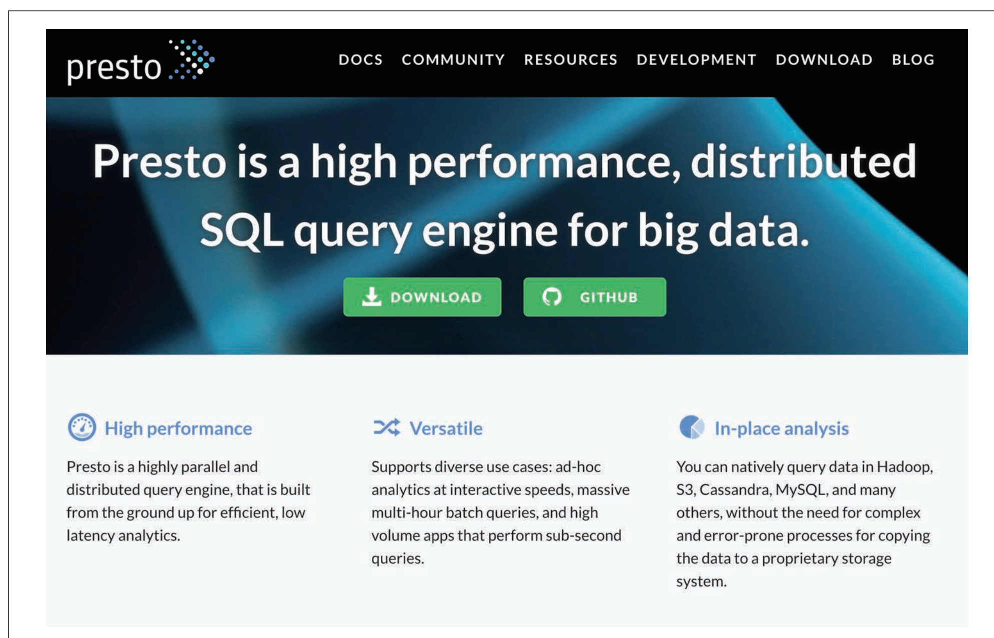


图 1-5：Presto 官方网站的主页

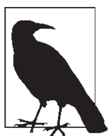
1.4.2 文档

Presto 的详细文档作为代码库的一部分维护，可以从官方网站上获取到。该文档既包括概览，又包括 SQL 支持、函数、运算符、连接器、配置等方面的详细参考信息，还提供包含最新修改细节的发布记录。你可以从 Presto 文档开始。

1.4.3 社区交流

你可以加入面向新手、进阶用户和专家用户，以及 Presto 代码贡献者和维护者的社区聊天室，Presto 社区非常乐于帮助参与者，成员们每天都在积极地互相协作。

你可以先加入 general 频道，然后查看专注于 bug 分类、版本发布和开发等多种话题的无数其他频道。



你总是可以在社区聊天室找到马特、曼弗雷德和马丁。我们非常乐意听取你的反馈。

1.4.4 源代码、许可证和版本

Presto 是在 Apache License v2 协议下的开源项目，其源代码在 Git 仓库中，可参见 GitHub 网站的 prestosql/presto 页面。

位于 GitHub 网站 Presto 页面的 prestosql 组织包含许多与项目相关的其他仓库，比如官方网站、客户端、其他组件的源代码或用于贡献者许可证管理的仓库。

Presto 是一个活跃且经常发布新版本的开源项目。使用最新版本时，你可以获得最新特性、bug 修复和性能提升。在编写本书时 Presto 的最新版本是 330，所以本书内容也使用和参考了这一版本。如果你使用一个更新版本的 Presto，它的工作原理应该能和本书中描述的一样。尽管你不太可能遇到问题，但对于任何修改，参考发布记录和文档的内容仍非常重要。

1.4.5 贡献

如前所述，Presto 是一个社区驱动的开源项目，我们欢迎并鼓励你为其贡献自己的力量。这个项目的社区聊天室非常活跃，你可以随时向代码提交者和其他开发者寻求帮助。

下面是在贡献代码之前需要完成的一些任务。

- 查看文档中的开发者导引章节。
- 从 README 文件中学习从源代码构建项目的方法。
- 在官方网站的社区（Community）页面找到研究论文的链接，并阅读研究论文。

- 在同一个页面上浏览行为守则（Code of Conduct）。
- 找到一个标记为 good first issue 的问题。
- 签署贡献者授权协议（contributor license agreement，CLA）。

Presto 项目持续接受各种复杂度的贡献：从小的文档提升、新的连接器或者其他插件，到深入 Presto 内部的改进等。

当然，社区欢迎你使用 Presto 或围绕 Presto 做的任何工作。这类工作当然也包括一些看似不相关的贡献，如编写博客文章、参加用户组的会议或研讨会、自己编写和维护插件（也许是用来连接你自己使用的一个数据库系统）等。

总之，我们鼓励你和 Presto 团队一起工作并参与进来，这个项目的成长和繁荣有赖于每个人的贡献。我们也时刻准备着帮助你。你可以做到！

1.4.6　本书资源

我们提供了一些有关本书的资源，如配置文件示例、SQL 查询、数据集等。

可以在本书图灵社区页面（http://ituring.cn/book/2768）找到这些资源，然后以压缩文件的方式下载。

1.4.7　鸢尾花数据集

在本书后面的章节中，你将遇到有关鸢尾花和鸢尾花数据集的查询示例和使用场景。该数据集广为人知，常用于数据科学中有关分类的例子。

该数据集包含一个简单的表，该表有 150 条记录，以列的形式提供了萼片长度、萼片宽度、花瓣长度、花瓣宽度和物种的值。

这个数据集非常小，可以让用户方便地测试和执行查询，并完成一系列不同的分析任务。因此，该数据集非常适合学习，你可以从维基百科的对应页面上了解有关此数据集的更多信息。

本书的仓库包含了一个名为 iris-data-set 的目录。该目录中存放着以逗号分隔值（CSV）格式保存的数据，还包含了用于创建表并将数据插入的 SQL 文件。读完第 2 章和 3.1 节后，以下指令将非常容易操作。

要使用鸢尾花数据集，你可以先复制 etc/catalog/memory.properties 文件到 Presto 的安装路径下并重启 Presto。

现在，你就可以使用 Presto CLI 将数据导入 `iris` 表中。此时，`iris` 表存在于 `default` schema 下，`default` schema 又存在于 `memory` catalog 下。

```
$ presto -f iris-data-set/iris-data-set.sql
USE
CREATE TABLE
INSERT: 150 rows
```

确认数据可以查询：

```
$ presto --execute 'SELECT * FROM memory.default.iris;'
"5.1","3.5","1.4","0.2","setosa"
"4.9","3.0","1.4","0.2","setosa"
"4.7","3.2","1.3","0.2","setosa"
...
```

此外，你还可以使用任何可以连接到 Presto 的 SQL 管理工具来运行查询，如 3.2 节介绍的 Java 数据库连接（JDBC）驱动。

第 8 章和第 9 章包含一些使用此数据集的查询案例，6.8 节提供了有关内存连接器的信息。

1.4.8 航班数据集

与鸢尾花数据集类似，本书后面还将介绍航班数据集的查询案例和使用场景。这一数据集比鸢尾花数据集复杂，它包含关于航空公司、机场及其他信息的多个查询表，也包含有关特定航班的业务数据。这使得此数据集适用于使用 Join 操作的更复杂的查询和联邦查询（其中不同表位于不同的数据源）。

此数据集的数据来自美国联邦航空管理局（FAA），专用于分析。`flights` 表的 schema 比较大，表 1-1 展示了其中一部分列。

表1-1：可用列的子集

flightdate	airlineid	origin	dest	arrtime	deptime

数据集中的每一行表示从美国机场起飞或到达美国机场的航班。

1.5 Presto简史

Facebook 于 2008 年开源了 Hive，它之后成了 Apache Hive。Hive 在 Facebook 内部被广泛用于在其巨大的 Apache Hadoop 集群上执行 HDFS 上的数据分析任务。

Facebook 的数据分析师使用 Hive 在大型数据仓库上执行交互式查询。在 Facebook 开发出 Presto 之前，所有的分析师都依赖于 Hive，但 Hive 并不适用于 Facebook 这种大数据规模下的交互式查询。2012 年，Facebook 的 Hive 数据仓库已经有 250PB 了，它每天需要处理数百个用户的成千上万条查询。在 Facebook 内部，Hive 逐渐到达极限，并且它无法查询其他数据源。

Facebook 从零开始设计出了 Presto，以便在其数据规模下快速地执行查询。Presto 不是创建一个新系统并将数据迁移过去，而是通过可插拔的连接器系统，在数据存储的本地读取数据。Hive 连接器是 Presto 最早的连接器之一（参见 6.4 节），它直接查询存储在 Hive 数据仓库的数据。

2012 年，4 个 Facebook 工程师开始研发 Presto，以满足 Facebook 在分析场景中对性能、扩展性和延伸性上的需求。Presto 最初就是要构建为开源项目。在 2013 年年初，Presto 的初始版本在 Facebook 内部生产环境上线。同年秋天，Facebook 正式开源 Presto。鉴于 Presto 在 Facebook 内部取得的成功，许多其他大型互联网公司也开始使用它，如 Netflix、LinkedIn、Treasure Data 等，并还有更多的公司紧随其后。

2015 年，Teradata 声明将向 Presto 投入多达 20 个工程师来贡献代码，他们专注于添加企业特性，如安全提升和生态系统工具集成。2015 年晚些时候，Amazon 开始在 AWS Elastic MapReduce（EMR）系统中添加 Presto。2016 年，Amazon 发布了以 Presto 作为主要基石的 Athena 服务。2017 年，Starburst 横空出世，它是一家专门推广 Presto 的公司。

2018 年末，Presto 的创始人离开 Facebook，成立了 Presto 软件基金会，以使该项目保持协作与独立。自那时起，Presto 的创新和增长速度都更上一层楼。

今天，Presto 社区欣欣向荣，许多知名公司大规模地使用它。这个项目现在由来自世界各地许多公司的开发者和贡献者组成的繁荣社区维护，这些公司包括阿里巴巴集团、Amazon、Appian、Gett、Google、Facebook、Hulu、Line、LinkedIn、Lyft、Motorola、Qubole、Red Hat、Salesforce、Starburst、Twitter、Uber、Varada、Walmart 和 Zoho。

1.6 小结

本章介绍了 Presto 以及它的一些特性，还探索了其可能的使用场景。

在第 2 章中，你将安装 Presto 并运行它，将它连接到一个数据源，并了解如何查询数据。

第 2 章

安装和配置Presto

在第 1 章中，你了解了 Presto 及其可能的使用场景，现在可以开始尝试了。在本章中，你将了解如何安装 Presto、配置数据源并查询数据。

2.1　使用Docker容器尝试Presto

Presto 项目提供了 Docker 容器，便于启动配置好的 Presto 演示环境，用于初步了解并探索 Presto。

要在 Docker 里运行 Presto，需要先在机器上安装 Docker，这可以从 Docker 的官方网站下载，或使用操作系统所提供的包管理系统。

使用 `docker` 命令下载容器镜像，将其命名为 `presto-trial` 并保存在本地，然后在后台运行它：

```
docker run -d --name presto-trial prestosql/presto
```

现在让我们连接到容器内，并使用 Presto 命令行界面（CLI）执行 `presto` 命令。这会连接到同一容器里运行着的 Presto 服务器。接下来，在命令提示行中可以执行一个在 TPC-H 基准测试数据表上的查询。

```
$ docker exec -it presto-trial presto
presto> select count(*) from tpch.sf1.nation;
 _col0
-------
    25
(1 row)
```

```
Query 20181105_001601_00002_e6r6y, FINISHED, 1 node
Splits: 21 total, 21 done (100.00%)
0:06 [25 rows, 0B] [4 rows/s, 0B/s]
```



如果你尝试运行 Docker 并遇到了如下错误：`Query 20181217_115041_00000_i6juj failed: Presto server is still initializing`，请等待一段时间后重试上一条命令。

利用 SQL 知识，你可以进一步探索这一数据集。可以使用 `help` 命令了解 Presto CLI，更多有关使用 CLI 的信息，参见 3.1 节。

当你结束探索时，可以使用 `quit` 命令退出。

执行下列命令就可以停止并移除之前的容器：

```
$ docker stop presto-trial
presto-trial
$ docker rm presto-trial
presto-trial
```

如果想进一步试验，则可以使用同样的命令运行 Presto。如果试用结束并不再需要 Docker 镜像，则可以通过下列命令删除所有相关的 Docker 资源：

```
$ docker rmi prestosql/presto
Untagged: prestosql/presto:latest
...
Deleted: sha256:877b494a9f...
```

2.2 使用归档文件安装

使用 Docker 初试 Presto 后，可以选择在本地的工作站或服务器上安装 Presto。

Presto 可以运行在大多数现代 Linux 发行版和 macOS 上，它依赖 Java 虚拟机（JVM）和 Python 环境。

2.2.1 JVM

Presto 使用 Java 编写并要求你的系统内安装有 JVM。Presto 需要 Java 的长期支持版本 Java 11，不支持更旧版本的 Java，而在更新版本的 Java 环境下它也许可以正常工作，但并未仔细测试。

下列命令确认 `java` 命令已经安装并且在 `PATH` 环境变量中：

```
$ java --version
openjdk 11.0.4 2019-07-16
```

```
OpenJDK Runtime Environment (build 11.0.4+11)
OpenJDK 64-Bit Server VM (build 11.0.4+11, mixed mode, sharing)
```

若没有安装 Java 11，Presto 就会启动失败。

2.2.2　Python

Presto 的启动器脚本依赖 Python 2.6 或更高版本。

下列命令确认 `python` 命令已经安装并且在 `PATH` 环境变量中：

```
$ python --version
Python 2.7.15+
```

2.2.3　安装

Presto 的二进制包使用 Maven 中心仓库分发。服务端应用程序可以通过一个 tar.gz 归档文件获得。

你可以通过以下链接获得可用的版本列表：

https://repo.maven.apache.org/maven2/io/prestosql/presto-server

先找到最大的版本号（代表最新发布的版本），进入对应的文件夹并下载 tar.gz 文件。也可以使用命令行工具下载归档文件。例如，使用 `wget` 命令下载 330 版本：

```
$ wget https://repo.maven.apache.org/maven2/\
  io/prestosql/presto-server/330/presto-server-330.tar.gz
```

然后提取归档文件：

```
$ tar xvzf presto-server-*.tar.gz
```

提取过程会创建一个顶级目录，其名称和归档文件名相同，但不带扩展名。这一目录被称为**安装**目录。

安装目录包含下列子目录。

lib

　包含组成 Presto 服务端及其全部依赖关系的 Java 归档文件（JAR）。

plugins

　包含 Presto 插件及其依赖关系，各个插件分开存放在不同的目录下。Presto 默认已经包含许多插件，你也可以添加第三方插件。Presto 允许多种可插拔的组件，如连接器、函数和安全访问控制等与其集成。

bin

包含 Presto 的启动脚本。这些脚本用于启动、停止、重启、终结 Presto 进程，以及获得运行该进程的状态。你可以在 5.6 节中了解更多有关这些脚本的使用。

etc

配置文件所在的目录。此目录由用户创建并提供 Presto 必需的配置文件。你可以在 5.1 节中找到更多有关配置选项的信息。

var

存放日志信息的数据目录。Presto 服务端第一次启动时会创建此目录，它默认情况下位于安装目录下。我们推荐将其配置为安装目录之外的位置，以便在升级过程中保留这些数据。

2.2.4　配置

启动 Presto 前，需要先提供一些配置文件：

- Presto 日志配置
- Presto 节点配置
- JVM 配置

默认情况下，配置文件应存放在安装目录下的 etc 目录里。

除了 JVM 配置之外，其他配置文件的格式遵循 Java properties 标准。简单来说，在 Java properties 中，每个配置参数都以 `key=value` 字符串对的格式存储为一行。

你需要在 2.2.3 节创建的 Presto 安装目录中创建这些基本的配置文件，或在本书的 Git 仓库（参见 1.4.6 节）中找到可以直接使用的配置文件。下面是三个配置文件的内容。

etc/config.properties：

```
coordinator=true
node-scheduler.include-coordinator=true
http-server.http.port=8080
query.max-memory=5GB
query.max-memory-per-node=1GB
query.max-total-memory-per-node=2GB
discovery-server.enabled=true
discovery.uri=http://localhost:8080
```

etc/node.properties：

```
node.environment=demo
```

etc/jvm.config：

```
-server
```

```
-Xmx4G
-XX:+UseG1GC
-XX:G1HeapRegionSize=32M
-XX:+UseGCOverheadLimit
-XX:+ExplicitGCInvokesConcurrent
-XX:+HeapDumpOnOutOfMemoryError
-XX:+ExitOnOutOfMemoryError
-Djdk.nio.maxCachedBufferSize=2000000
-Djdk.attach.allowAttachSelf=true
```

上述的配置文件就绪后，Presto 就可以启动了。可以在第 5 章中找到这些配置文件更详细的描述。

2.3 添加数据源

虽然已经安装好了 Presto，但还不能就这样启动它。毕竟，你想要在 Presto 里查询一些外部数据，这需要你将一个数据源配置为 catalog。

Presto 的 catalog 定义了用户可用的数据源。数据访问是由 Presto 连接器执行的，该连接器由 `connector.name` 属性在 catalog 中配置。catalog 将数据源中的所有 schema 和表暴露给 Presto。

例如，Hive 连接器将每个 Hive 数据库映射到一个 schema。如果想要使用 Hive 连接器将一个名为 `web` 的 Hive 数据库（其中包含一张 `clicks` 表）暴露为名为 `sitehive` 的 catalog，那么我们需要在 catalog 文件中指定使用 Hive 连接器。你可以使用完全限定名语法 `catalog.schema.table` 来访问 catalog，本例中就是 `sitehive.web.clicks`。

可以通过在 etc/catalog 目录下创建一个 catalog 属性文件来注册 catalog。该文件的名称就是 catalog 在系统中的名称。比如，你创建了 etc/catalog/cdh-hadoop.properties、etc/catalog/sales.properties、etc/catalog/web-traffic.properties 和 etc/catalog/mysql-dev.properties 等 catalog 属性文件，则这些 catalog 在 Presto 里就是 `cdh-hadoop`、`sales`、`web-traffic` 和 `mysql-dev`。

可以使用 TPC-H 连接器探索 Presto。TPC-H 连接器内置于 Presto 中并提供了一系列的 schema 用于支持 TPC-H。你可以在 6.3 节中了解更多信息。

要配置 TPC-H 连接器，需要创建一个 catalog 属性文件 etc/catalog/tpch.properties，并将连接器配置为 `tpch`：

```
connector.name=tpch
```

每个 catalog 文件都必须有 `connector.name` 属性。额外的属性由对应 Presto 连接器的实现决定。这些内容在 Presto 的文档中都有介绍，你也可以在第 6 章和第 7 章中了解更多信息。

本书仓库包含了一系列 catalog 文件，来帮助你学习 Presto。

2.4 运行Presto

现在你终于真正准备好启动 Presto 了。安装目录下包含了启动脚本，你可以用它们来启动 Presto：

```
$ bin/launcher run
```

`run` 命令将 Presto 启动为前台进程，Presto 的日志和其他输出将写入标准输出流（`stdout`）和标准错误流（`stderr`）。成功启动后会有一条日志，你一段时间后应该可以看到以下输出：

```
INFO        main io.prestosql.server.PrestoServer ======== SERVER STARTED
```

在前台运行 Presto 有助于测试和快速检验进程能否正确启动，以及它是否使用了期望的配置。可以使用 Ctrl-C 组合键结束服务器进程。

5.6 节和 5.3 节分别介绍了有关启动器脚本和日志的更多信息。

2.5 小结

本章的内容让你了解到安装和启动 Presto 非常容易。现在可以启动它并将其投入使用了。

在第 3 章中，你将了解如何与 Presto 交互并用其查询配置好的数据源。你也可以直接跳到第 6 章和第 7 章来学习多种其他的连接器，并为你自己添加更多的 catalog。

第3章

使用Presto

祝贺你！在前面的章节里，我们向你介绍了 Presto，你学习了如何安装、配置和启动它，下面可以开始使用它了。

3.1 Presto CLI

Presto CLI 提供了一个基于终端的交互式 shell。你可以通过它运行查询并与 Presto 服务端交互来查看元数据。

3.1.1 使用入门

正如 Presto 服务端一样，Presto CLI 也通过 Maven 中央仓库分发二进制包。这个 CLI 应用程序以可执行 JAR 包的形式提供，你可以像普通 UNIX 可执行程序一样使用它。

你可以从如下链接查看可用版本的列表：https://repo.maven.apache.org/maven2/io/prestosql/presto-cli。

找到与你使用的 Presto 服务端版本相一致的 CLI 版本，从版本目录中下载对应的 *-executable.jar 文件，然后将其重命名为 presto。下列命令展示了如何使用 `wget` 完成上述步骤并安装 330 版本：

```
$ wget -O presto \
https://repo.maven.apache.org/maven2/\
io/prestosql/presto-cli/330/presto-cli-330-executable.jar
```

你需要确保文件具有可执行权限。为了方便使用，将其放置在 PATH 中。例如，你可以将其复制到 ~/bin，并将此目录添加到 PATH 中。

```
$ chmod +x presto
$ mv presto &#x7e;/bin
$ export PATH=~/bin/:$PATH
```

下面可以运行 Presto CLI 并确认其版本：

```
$ presto --version
Presto CLI 330
```

可以使用 help 选项查看所有可用的选项和命令的文档。

```
$ presto --help
```

开始使用 CLI 之前，你需要决定与哪个 Presto 服务端交互。默认情况下，CLI 连接到运行在 http://localhost:8080 上的 Presto 服务端。如果 Presto 服务端在本地运行（出于测试或开发目的），或者你通过 SSH 连接到了服务器，又或者你使用 exec 命令连接到安装了 CLI 的 Docker 容器中，则可以直接执行以下命令：

```
$ presto
presto>
```

如果 Presto 服务端运行在另一台服务器上，则需要手动指定 URL：

```
$ presto --server https://presto.example.com:8080
presto>
```

presto> 命令行提示符表明你在使用交互式控制台访问 Presto 服务端。键入 help 命令来获取可用命令的列表：

```
presto> help
Supported commands:
QUIT
EXPLAIN [ ( option [, ...] ) ] <query>
    options: FORMAT { TEXT | GRAPHVIZ | JSON }
             TYPE { LOGICAL | DISTRIBUTED | VALIDATE | IO }
DESCRIBE <table>
SHOW COLUMNS FROM <table>
SHOW FUNCTIONS
SHOW CATALOGS [LIKE <pattern>]
SHOW SCHEMAS [FROM <catalog>] [LIKE <pattern>]
SHOW TABLES [FROM <schema>] [LIKE <pattern>]
USE [<catalog>.]<schema>
```

CLI 中的大多数命令，尤其是所有 SQL 语句，需要以分号结尾。在 Presto 上使用 SQL 的更多信息，参见 3.6 节。现在，你只需完成一些简单的事情来入门即可。

首先，你可以检查哪些数据源被配置为 catalog。你至少会看到内部元数据 catalog——`system`。在我们的案例中，你还会看到 `tpch` catalog：

```
presto> SHOW CATALOGS;
 Catalog
---------
 system
 tpch
(2 rows)

Query 20191212_185850_00001_etmtk, FINISHED, 1 node
Splits: 19 total, 19 done (100.00%)
0:01 [0 rows, 0B] [0 rows/s, 0B/s]
```

你可以很容易地显示可用 schema 及其中的表。每次执行 Presto 查询时，查询处理过程的统计信息都会和查询结果一起返回。你可以在前面的代码中看到统计信息，但以下代码会省略它：

```
presto> SHOW SCHEMAS FROM tpch;
       Schema
--------------------
 information_schema
 sf1
 sf100
 sf1000
 sf10000
 sf100000
 sf300
 sf3000
 sf30000
 tiny
(10 rows)

presto> SHOW TABLES FROM tpch.sf1;
  Table
----------
 customer
 lineitem
 nation
 orders
 part
 partsupp
 region
 supplier
(8 rows)
```

现在你准备好查询实际的数据了：

```
presto> SELECT count(name) FROM tpch.sf1.nation;
 _col0
-------
    25
(1 row)
```

此外还可以选择一个 schema 作为当前 schema，之后就可以从查询中省略限定符了：

```
presto> USE tpch.sf1;
USE
presto:sf1> SELECT count(name) FROM nation:
```

如果你知道将在某一 schema 下工作，则可以在启动 CLI 时指定它：

```
$ presto --catalog tpch --schema sf1
```

现在，你能够利用你所有的 SQL 知识和 Presto 的强大能力查询已配置的数据源了。

要退出 CLI，只需键入 `quit` 或 `exit` 命令，或按 Ctrl-D 组合键。

3.1.2　分页

默认情况下，查询的结果将使用精心配置好的 `less` 程序分页。你可以通过将环境变量 `PRESTO_PAGER` 设置为另一个程序的名称（如 `more`）来覆盖这一行为，或将其置为空来彻底禁用分页。

3.1.3　命令历史

Presto CLI 会保留之前运行过的命令的历史。你可以使用上下箭头键来滚动浏览这些历史，也可以使用 Ctrl-S 组合键和 Ctrl-R 组合键来查询历史。要再次执行一条查询，只需按回车键即可。

默认情况下，Presto 的命令历史保存在 ~/.presto_history 文件中，你可以使用 `PRESTO_HISTORY_FILE` 环境变量来修改默认值。

3.1.4　额外诊断

Presto CLI 提供了 `--debug` 选项来打开运行查询时的调试信息输出。

```
$ presto --debug

presto:sf1> SELECT count(*) FROM foo;
Query 20181103_201856_00022_te3wy failed:
 line 1:22: Table tpch.sf1.foo does not exist
io.prestosql.sql.analyzer.SemanticException:
 line 1:22: Table tpch.sf1.foo does not exist
...
at java.lang.Thread.run(Thread.java:748)
```

3.1.5　执行查询

你可以直接使用 `presto` 命令执行查询，并在查询执行完毕时退出 Presto CLI。这在你需要

编写执行多条查询的脚本，或使用其他系统实现更复杂的自动化工作流时很有用。以这种方式执行命令时，它会返回来自 Presto 的查询结果。

要使用 Presto CLI 直接执行查询，你需要使用 `--execute` 选项。注意，在查询语句中你需要使用完全限定符来引用一个表（例如 `catalog.schema.table`）。

```
$ presto --execute 'SELECT nationkey, name, regionkey FROM tpch.sf1.nation LIMIT 5'
"0","ALGERIA","0"
"1","ARGENTINA","1"
"2","BRAZIL","1"
"3","CANADA","1"
"4","EGYPT","4"
```

你也可以使用 `--catalog` 和 `--schema` 选项来避免使用完全限定符：

```
$ presto --catalog tpch --schema sf1 \
  --execute 'select nationkey, name, regionkey from nation limit 5'
```

借助分号分隔不同的查询语句，你可以一次执行多个查询。

Presto CLI 也支持读入并执行文件中的命令和 SQL 查询，如 nations.sql 文件包含以下内容：

```
USE tpch.sf1;
SELECT name FROM nation;
```

你可以在 CLI 中使用 `-f` 选项执行该文件中的命令。Presto CLI 会将结果输出到命令行中，然后退出：

```
$ presto -f nations.sql
USE
"ALGERIA"
"ARGENTINA"
"BRAZIL"
"CANADA"
...
```

3.1.6 输出格式

Presto CLI 提供了 `--output-format` 选项来控制如何在非交互模式下显示输出，可用的选项有 `ALIGNED`、`VERTICAL`、`CSV`、`TSV`、`CSV_HEADER`、`TSV_HEADER` 和 `NULL`，默认值是 `CSV`。

3.1.7 忽略错误

Presto CLI 提供了 `--ignore-error` 选项，使用它你可以忽略执行文件中的查询时遇到的任何错误。默认行为是在遇到第一个错误时终止执行脚本。

3.2 Presto JDBC驱动

任何 Java 应用程序都可以通过 Java 数据库连接（JDBC）驱动连接到 Presto。JDBC 是一套标准的数据库访问 API，它提供了在关系数据库中查询、插入、删除和更新数据等的必要方法。许多运行在 JVM 上的客户端应用程序或服务端应用程序使用 JDBC 来访问底层的数据库，以实现数据库管理、报表和其他一些特性。通过 JDBC 驱动，所有这些应用程序都可以使用 Presto。

Presto 的 JDBC 驱动允许你连接到 Presto 并使用 SQL 语句与 Presto 交互。



如果你熟悉 JDBC 驱动的不同实现，就知道 Presto 的 JDBC 驱动是 Type 4 驱动，这仅仅意味着它直接与 Presto 原生接口通信。

JDBC 驱动可以让你使用许多强大的 SQL 客户端和数据库管理工具，比如开源工具 DBeaver 或 SQuirreL SQL 客户端等。基于 JDBC 的报告生成、仪表盘和分析工具也可以与 Presto 一起使用。

用这些工具连接 Presto 的步骤都很相似。

1. 下载 JDBC 驱动。
2. 将 JDBC 驱动放置在应用 classpath 的覆盖范围之内。
3. 配置 JDBC 驱动。
4. 配置 Presto 连接。
5. 连接 Presto 并开始使用。

例如，开源数据库管理工具 DBeaver 简化了上述过程。安装和启动 DBeaver 之后，遵循以下步骤。

1. 从 File（文件）菜单中选择 New（新建）。
2. 在 DBeaver 这部分中，选择 Database Connection（数据库连接），然后单击 Next（下一步）。
3. 在输入框中键入 `prestosql`，选择对应图标，然后单击 Next（下一步）。
4. 配置到 Presto 的连接，然后单击 Finish（完成）。注意，username（用户名）字段是**必填项**。在 Presto 的默认安装环境中你可以提供一个随机的用户名而无须认证。

现在你可以在左边的数据库导航器（Database Navigator）中看到这个连接，并可以检视 schema 和表，图 3-1 显示了一个例子。你也可以启动 SQL 编辑器，开始编写查询并用 Presto 工作。

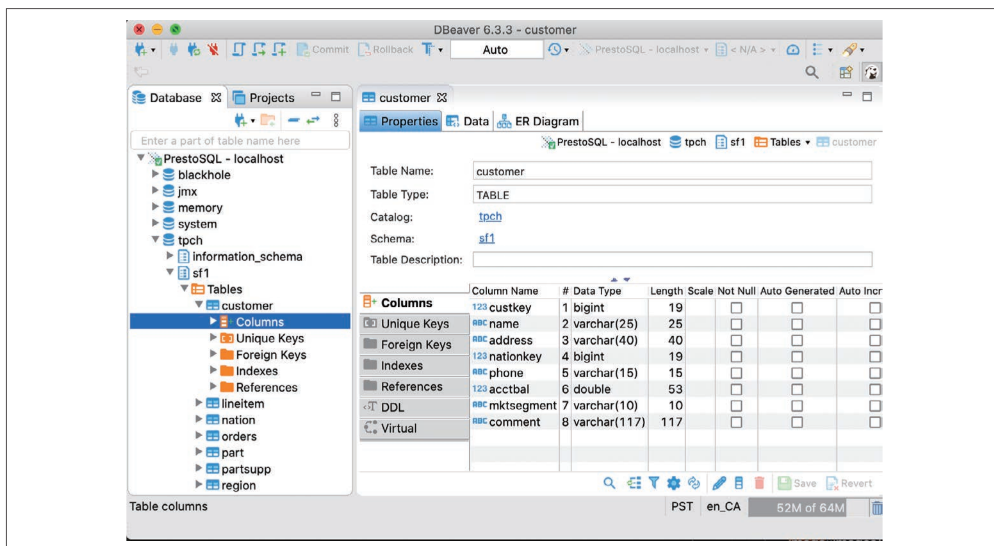


图 3-1：DBeaver 用户界面（显示 tpch.sf1.customer 表的列信息）

SQuirreL SQL 客户端和许多其他工具具有相似的过程。有些步骤需要更多手动操作，如下载 JDBC 驱动、配置数据库驱动和连接等。下面来看一下相关细节。

3.2.1　下载和注册驱动

Presto 的 JDBC 驱动使用 Maven 中央仓库分发，该驱动程序被打包成 JAR 文件。

你可以从如下链接获得可用版本的列表：https://repo.maven.apache.org/maven2/io/prestosql/presto-jdbc。

找到代表最新发布版本的最大的数字，进入对应目录并下载 .jar 文件。你也可以在命令行里下载它，例如，使用 wget 命令下载版本 330：

```
$ wget https://repo.maven.apache.org/maven2/\
io/prestosql/presto-jdbc/330/presto-jdbc-330.jar
```

要在应用程序中使用 Presto 的 JDBC 驱动，你先要将它加入 Java 应用程序的 classpath 中。不同的应用程序有不同的 classpath，但像 SQuirreL SQL 客户端一样，它们通常使用一个名为 lib 的目录。有些应用程序提供一个对话框来将库文件添加到 classpath，也可以手动将文件复制到指定位置。

加载驱动通常需要重启应用程序。

你现在可以注册驱动了。在 SquirreL SQL 客户端中，你可以在用户界面左边的驱动（Drivers）标签页单击 + 按钮来创建一个新驱动。

在配置驱动时，你需要确保配置了以下参数。

- 类名称：`io.prestosql.jdbc.PrestoDriver`
- JDBC URL 示例：jdbc:presto://host:port/catalog/schema
- 名称：`Presto`

要让驱动工作，只有类名称、JDBC URL 和在 classpath 中的 JAR 包是必填项。其他参数是可选的，并且因应用程序不同而有所不同。

3.2.2 创建到Presto的连接

注册好驱动，且启动并运行 Presto 之后，你现在可以将其连接到应用程序了。

在 SQuirreL SQL 客户端中，连接配置叫作**别名**（alias）。你可以在用户界面左边的别名（Alias）标签页上单击 + 按钮，并使用以下参数来新建别名。

名称

描述 Presto 连接的名称。如果你要连接到多个 Presto 实例，或不同的 schema 和数据库，好的名称就更重要了。

驱动

选择你之前创建的 Presto 驱动。

URL

连接到 Presto 的 JDBC URL 采用模板 jdbc:presto://host:port/catalog/schema，其中 catalog 和 schema 的值是可选的。要连接到你之前在本地安装并运行的 Presto 服务（运行在 http://localhost:8080 上），可以使用 jdbc:presto://localhost:8080 作为 JDBC URL。`host` 参数指定了 Presto 协调器运行的主机地址，它也是你通过 Presto CLI 进行连接时使用的主机名。`port` 参数指定的是对应主机上 Presto 的 HTTP 端口。可选的 `catalog` 和 `schema` 参数用于在指定 catalog 和 schema 的情况下建立连接。如果你指定了 catalog 和 schema，查询语句中的表名称就无须使用完全限定符。

用户名

即使 Presto 上没有配置权限认证，用户名也是**必填项**，这使得 Presto 可以报告查询的发起人。

密码

密码与用户相关联并用于认证。Presto 的默认安装没有配置权限认证，因此无须密码。

Presto 的 JDBC 驱动可以接收更多参数作为属性，这样应用程序便可以决定提供哪些值。作为连接配置的一部分，DBeaver 和 SQuirreL SQL 客户端都包含一个用户界面来指定属性。

SSL

是否开启连接的 SSL 加密，可选值：`true` 或者 `false`。

SSLTrustStorePath

SSL 信任库（truststore）的路径。

SSLTrustStorePassword

SSL 信任库的密码。

`user` 和 `password`

等价于用户名和密码参数。

applicationNamePrefix

该属性用于在 Presto 处标识应用程序，并设定 Presto 查询的源名称。源名称会显示在 Presto Web UI 中，以便管理员查看查询是从哪里发起的。此外，该属性可以与资源组配合使用。在资源组中，你可以使用 `ApplicationName` 属性来决定 Presto 如何分配资源。12.8 节对此进行了讨论。

Presto 的文档提供了 Presto JDBC 驱动可用参数的完整列表，参见 1.4.2 节。

一旦你配置好了连接，就可以使用它连接到 Presto，来查询 Presto 自身以及所有已配置的 schema 和数据库。查询执行、报告生成以及其他功能可用的特定特性，根据连接到 Presto 的应用程序不同，也会有所不同。图 3-2 展示了在 SQuirreL SQL 客户端中成功连接到 Presto 的一个连接，以及一些查询示例和结果集。

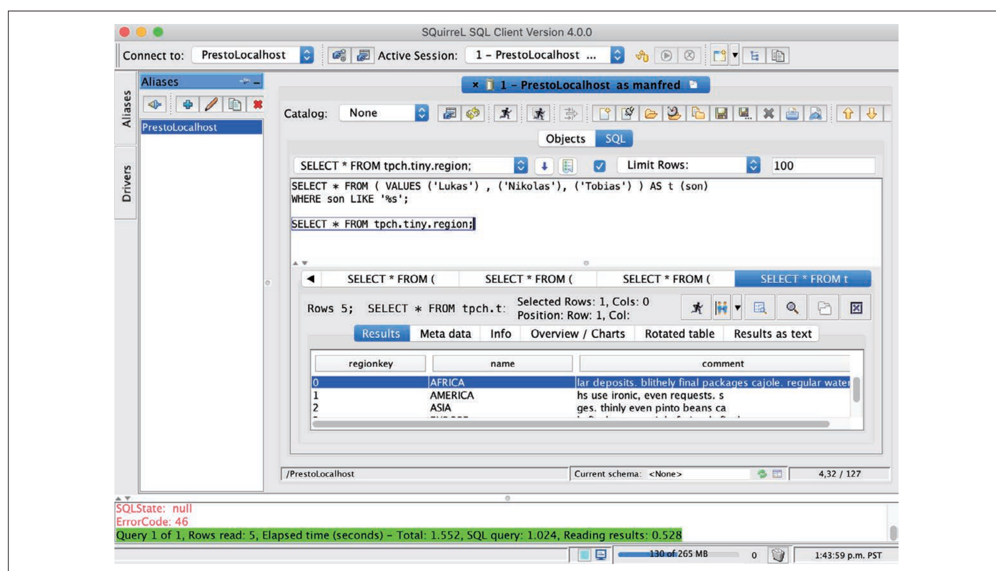


图 3-2：SQuirreL SQL 客户端用户界面显示若干查询以及结果集

3.3 Presto与ODBC

与使用 JDBC 驱动（参见 3.2 节）连接到 Presto 相似，开放数据库连接（ODBC）允许任何支持 ODBC 的应用程序连接到 Presto，并为典型的基于 C 的应用程序提供了一套 API。

目前 Presto 没有开源的 ODBC 驱动可用，但你可以从 Starburst 和 Simba 购买商业授权的驱动。

有了 ODBC 驱动支持，就可以使用数据库管理、商业智能以及报告与分析在内的诸多热门应用程序，如 Microsoft Power BI、Tableau、SAS、Quest Toad 等。ODBC 也支持 Microsoft Excel。

3.4 客户端库

除了由 Presto 团队直接维护的 Presto CLI 和 JDBC 驱动，Presto 社区的很多成员也为 Presto 创建了很多客户端库。

你可以找到为 Python、C、Go、Node.js、R、Ruby 等语言开发的库。Presto 网站（参见 1.4.1 节）维护了一个客户端列表。

这些库使你可以将 Presto 与这些语言生态系统中的应用程序集成起来，包括你自己编写的应用程序。

3.5 Presto Web UI

Presto 服务端提供了一个 Web 界面，通常叫作 Presto Web UI。如图 3-3 所示，Presto Web UI 提供了 Presto 服务端及其查询处理的详细信息。

Presto Web UI 可以借助与 Presto 服务端相同的 HTTP 端口号，使用相同的地址访问。默认端口是 8080，则一个可能的访问地址是 http://presto.example.com:8080。因此，在本地的安装环境下，可以在 http://localhost:8080 处访问 Web UI。

主仪表盘显示了 Presto 利用率的细节和查询列表。你还可以通过 UI 获得更进一步的信息。这些信息对操作 Presto 和管理运行中的查询具有极大的价值。

使用 Web UI 有助于监控 Presto 和性能调优，12.1 节有详细介绍。新手主要可以用它来确认服务端是否正常启动和运行，并正在处理查询。

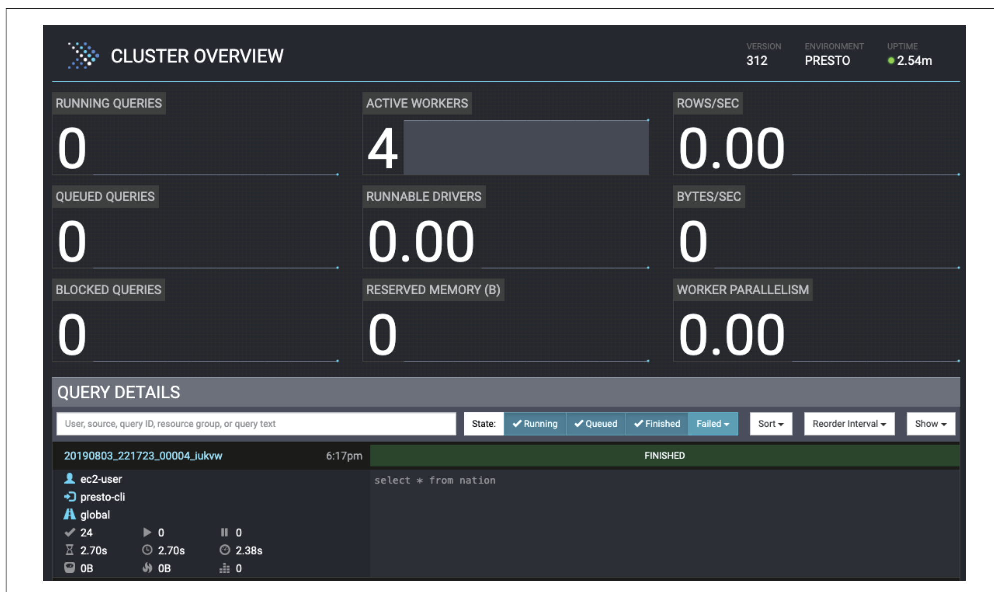


图 3-3：显示有关集群总体信息的 Presto Web UI

3.6 使用Presto执行SQL

Presto 是一个兼容 ANSI SQL 的查询引擎，可以让你使用相同的 SQL 语句、函数和运算符来查询和操作所有可以连接到的数据源。

Presto 致力于成为一个与现有 SQL 标准兼容的系统，其主要设计原则之一是既不发明一套新的类 SQL 查询语言，也不偏离 SQL 标准太远。它所有的新功能和语言特性都试图与标准兼容。

只有在标准没有定义某一等价的功能时，Presto 才会考虑扩展 SQL 特性。即使如此，Presto 在设计新的语言特性时也会非常小心。标准 SQL 和其他 SQL 实现里的相似特性会被考虑在内，以判断其是否有可能在未来标准化。



在少数 SQL 标准没有定义等价功能的情况下，Presto 会扩展标准，一个突出的例子是 lambda 表达式，参见 9.20 节。

Presto 没有将自己绑定在某一个特定的 SQL 标准版本上，相反，它将 SQL 标准看作活动的文档，并且非常重视最新版本的标准。另外，Presto 还未完全实现 SQL 标准所定义的所有必要特性。一条规定是，如果现存的某个功能被发现与标准不兼容，就会被弃用并很快替换为兼容标准的实现。

如前所述，你可以使用 Presto CLI 以及数据库管理工具（与 JDBC 或 ODBC 连接）来查询 Presto。

3.6.1 概念

Presto 使你可以使用 SQL 访问外部**数据源**，如关系数据库、键值存储和对象存储等。理解以下 Presto 概念非常重要。

连接器

使 Presto 适配一个数据源。每一个 catalog 对应于一个特定的连接器。

catalog

定义连接到一个数据源的细节。它包含了 schema 并配置了一个连接器来使用。

schema

组织表的一种方式。catalog 和 schema 一起定义了一个集合的表，这些表可以查询。

表

表是无序的行的集合。这些行内容被组织成带有数据类型的有名称的列。

3.6.2 入门案例

这一节给出了支持的 SQL 和 Presto 语句的一个简短概览，更多信息参见第 8 章和第 9 章。

`system` catalog 包含了 Presto 的元数据。你可以使用特定的语句来查询元数据，比如获得可用的 catalog、schema、信息 schema、表等。

使用下列语句来列出所有的 catalog：

```
SHOW CATALOGS;
 Catalog
 ---------
 system
 tpch
 (2 rows)
```

使用下列语句显示 `tpch` catalog 下所有的 schema：

```
SHOW SCHEMAS FROM tpch;
       Schema
 ------------------
 information_schema
 sf1
 sf100
 sf1000
 sf10000
```

```
 sf100000
 sf300
 sf3000
 sf30000
 tiny
(10 rows)
```

使用下列语句列出 `sf1` catalog 下所有的表：

```
SHOW TABLES FROM tpch.sf1;
  Table
 ---------
 customer
 lineitem
 nation
 orders
 part
 partsupp
 region
 supplier
(8 rows)
```

使用下列语句了解 `region` 表的数据类型：

```
DESCRIBE tpch.sf1.region;
  Column   |     Type     | Extra | Comment
-----------+--------------+-------+---------
 regionkey | bigint       |       |
 name      | varchar(25)  |       |
 comment   | varchar(152) |       |
(3 rows)
```

除此之外，还有一些有用的语句，如 `USE` 和 `SHOW FUNCTIONS` 等。8.1 节包含了有关 system catalog 和 Presto 语句的更多信息。

知道有哪些 catalog、schema 和表后，就可以使用标准 SQL 来查询数据了。

可以查看有哪些 `region`：

```
SELECT name FROM tpch.sf1.region;
    name
 ------------
 AFRICA
 AMERICA
 ASIA
 EUROPE
 MIDDLE EAST
(5 rows)
```

可以返回数据的一个子集并对列表排序：

```
SELECT name
FROM tpch.sf1.region
WHERE name like 'A%'
ORDER BY name DESC;
 name
---------
ASIA
AMERICA
AFRICA
(3 rows)
```

也支持连接多个表以及其他 SQL 标准中的功能：

```
SELECT nation.name AS nation, region.name AS region
FROM tpch.sf1.region, tpch.sf1.nation
WHERE region.regionkey = nation.regionkey
AND region.name LIKE 'AFRICA';
   nation   | region
------------+--------
 MOZAMBIQUE | AFRICA
 MOROCCO    | AFRICA
 KENYA      | AFRICA
 ETHIOPIA   | AFRICA
 ALGERIA    | AFRICA
(5 rows)
```

Presto 支持 || 作为字符串连接运算符。你也可以使用算术运算符，如 + 和 -。

可以将前面的查询改为使用 JOIN，并将返回的字符串结果连接成一个字段：

```
SELECT nation.name || ' / ' || region.name AS Location
FROM tpch.sf1.region JOIN tpch.sf1.nation
ON region.regionkey = nation.regionkey
AND region.name LIKE 'AFRICA';
      Location
 -------------------
 MOZAMBIQUE / AFRICA
 MOROCCO / AFRICA
 KENYA / AFRICA
 ETHIOPIA / AFRICA
 ALGERIA / AFRICA
(5 rows)
```

除了这些运算符，Presto 还支持各种各样的函数，它们复杂程度各不相同。在 Presto 中，可以使用 SHOW FUNCTIONS 语句显示这些函数的列表。

下面是一个简单的示例，它可以计算所有订单的平均价格，并输出四舍五入之后的整数值：

```
SELECT round(avg(totalprice)) AS average_price
FROM tpch.sf1.orders;
 average_price
 --------------
     151220.0
(1 row)
```

关于 SQL 使用的更详细信息，参见 Presto 的文档和本书第 8 章。Presto 官方网站上有函数和运算符的相关信息，第 9 章也提供了很好的概览和更多的示例。

3.7 小结

Presto 已可以启动并运行，你可以连接到一个数据源并使用 SQL 来查询它。你能够使用 Presto CLI 或者在其他应用程序中使用 JDBC 来连接 Presto。

有这一功能强大的组合在手，你已经准备好进一步研究了。在接下来的章节里，我们将学习如何进行大规模的生产环境部署、理解 Presto 的架构并了解 SQL 使用的细节。

第二部分

深入理解Presto

你已经初步了解了 Presto 及其多种使用场景，还安装并使用了它，下面可以深入学习 Presto 的底层设计了。

在本书的第二部分，你将学习 Presto 的内部工作原理，为在生产环境下安装、使用、运行以及调优 Presto 做准备。

这一部分还将详细讨论如何连接数据源，然后利用 Presto 对 SQL 语句、运算符、函数等的支持查询数据源。

第 4 章

Presto的架构

前面几章介绍了 Presto 及其初始安装和使用，本章来讨论 Presto 的架构。我们将深入介绍 Presto 的查询执行模型、查询计划以及基于代价的查询优化算法等相关概念。

首先本章从宏观上介绍 Presto 的架构组件。大致了解 Presto 的工作方式非常重要，特别是要自己安装和维护一个 Presto 集群（参见第 5 章）时。

然后本章会在讨论 Presto 的执行模型时更深入地介绍这些组件。这些知识是你诊断和调优慢查询（参见第 8 章）或向 Presto 开源项目贡献代码的基础。

4.1　集群中的协调器和工作节点

如果按照第 2 章介绍的方法安装并使用 Presto，它就只会运行在一台机器上，这种部署方式不具备可扩展性且性能受限。

Presto 是一个分布式 SQL 查询引擎，类似于**大规模并行处理**（massively parallel processing，MPP）数据库。Presto 通过在整个集群的服务器上分配处理任务来实现横向扩展，而非通过提高单台服务器性能来进行纵向扩展。这意味着，你可以通过添加更多的计算节点来获得更强大的处理能力。

基于这种架构，Presto 的查询引擎可以在集群内的计算节点上并行处理海量数据上的 SQL 查询。在每个计算节点上，Presto 都以单个服务端进程的形式运行。一个 Presto 集群由多个被配置为相互协作的 Presto 节点所组成。

图 4-1 是一个具有单个协调器和多个工作节点的 Presto 集群的概览。用户使用客户端程序

（如使用 JDBC 驱动的工具或 Presto CLI）连接到集群中的协调器。协调器之后协调工作节点访问数据源。

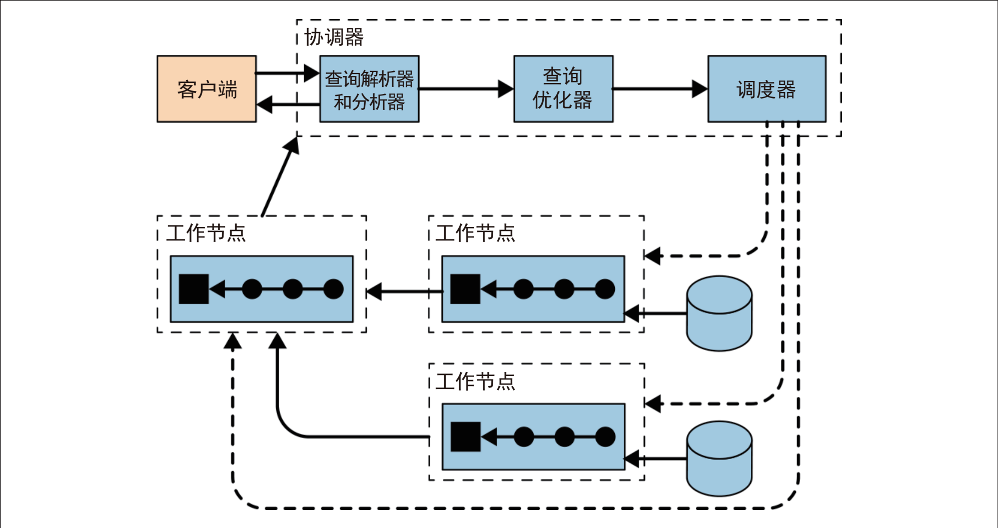


图 4-1：包含协调器和工作节点的 Presto 架构概览

协调器用于接受用户查询并管理**工作节点**以执行查询工作。

工作节点负责执行任务和处理数据。

协调器上通常会运行一个节点**发现服务**（discovery service），工作节点通过注册到此服务以加入集群。

客户端、协调器和工作节点之间的通信和数据传输完全通过基于 HTTP/HTTPS 的 RESTful API 调用。

图 4-2 展示了集群里协调器和工作节点（以及工作节点之间）是如何通信的。协调器为工作节点分配任务、更新状态并从工作节点获取顶层的结果集返回给用户。工作节点从数据源以及运行在其他节点上的上游任务中获取数据。

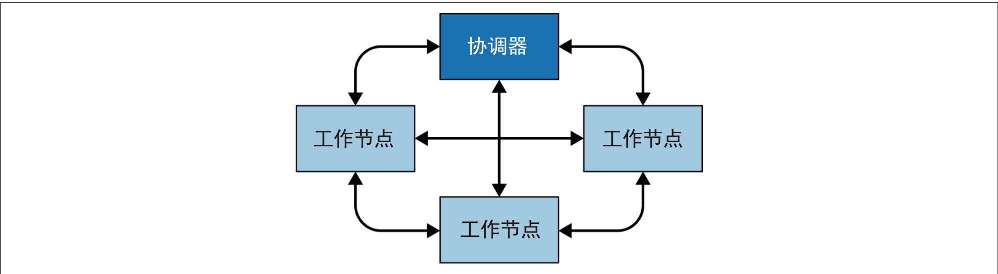


图 4-2：Presto 集群中协调器和工作节点之间的通信

4.2 协调器

Presto **协调器**负责接收用户 SQL 查询、解析查询语句、规划查询执行并管理工作节点。协调器是 Presto 集群的大脑，所有客户端应用程序都会连接它。用户可以通过 Presto CLI、使用 JDBC 或 ODBC 驱动的应用程序以及其他语言下可用的客户端库来与协调器交互。协调器接收客户端发送来的 SQL 语句（如 `SELECT` 查询）并执行。

一个 Presto 集群至少包含一个协调器，可能包含一个或多个工作节点。在开发和测试场景下，单个 Presto 实例可以同时承担协调器和工作节点的职责。

协调器会跟踪每个工作节点的动态，并协调查询任务的执行。对于一条查询，协调器将创建一个包含多个 Stage（阶段）的逻辑模型。

图 4-3 展示了客户端、协调器和工作节点之间的通信。

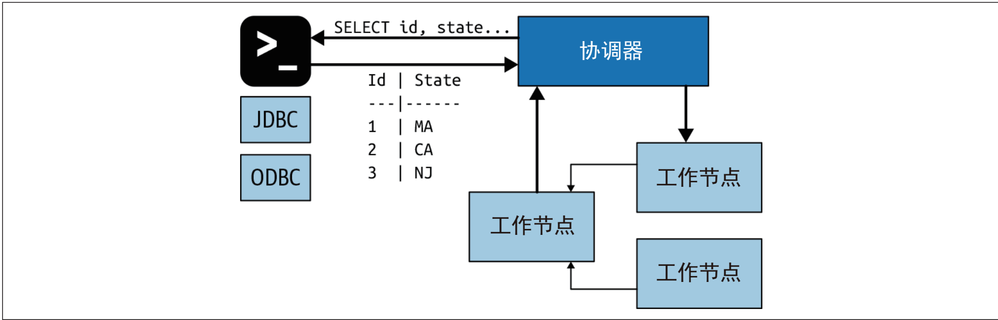


图 4-3：处理 SQL 语句时客户端、协调器和工作节点之间的通信

一旦接收到一条 SQL 语句，协调器就负责解析、分析、优化和调度查询任务在 Presto 工作节点上的执行。查询语句被翻译成一系列相连的任务（Task），这些任务被分发到各个工作节点上执行。在工作节点处理数据的同时，协调器会将结果抽取出来放到输出缓冲区中，并将缓冲区的内容暴露给客户端。一旦客户端读完输出缓冲区的内容，协调器就会代表客户端向工作节点请求更多的数据。另外，工作节点也在不断地与数据源交互并从中读取数据。最终，客户端不断地请求数据，工作节点则不断地从数据源读取数据并提供给客户端，直到查询执行结束。

协调器、工作节点和客户端基于 HTTP 进行通信。

4.3 节点发现服务

Presto 使用**节点发现服务**来发现集群中的所有节点。每个 Presto 实例在启动时都会注册到发现服务并定期发送心跳信号。这样一来，协调器就能够维护一个可用工作节点的最新列表，并用它来调度查询执行。

如果无法接收到一个工作节点的心跳信号，节点发现服务就会触发错误检测器，此后该工作节点不会再参与新的任务。

为了简化部署并避免运行额外的服务，Presto 的协调器通常会运行一个内嵌版的节点发现服务。因为它与 Presto 共享一个 HTTP 服务端，所以使用同一个端口。

因此，工作节点通常将节点发现服务的配置指向协调器的主机名和端口。

4.4 工作节点

工作节点是 Presto 集群中的一个服务端程序。它负责执行协调器分配的任务并处理数据。工作节点使用连接器从数据源获取数据，它们相互之间也会交换中间数据。它们将最终的结果数据发送给协调器，由协调器负责收集来自各个工作节点的结果并发送给客户端。

在安装时，工作节点需要知道集群的节点发现服务所在的主机名或 IP 地址。工作节点会在启动时将自己注册到节点发现服务上，以便协调器向其分配任务。

工作节点使用 HTTP 与其他工作节点和协调器通信。

图 4-4 展示了多个工作节点如何从数据源获取数据并合作处理数据，直到其中一个节点可以向协调器发送结果。

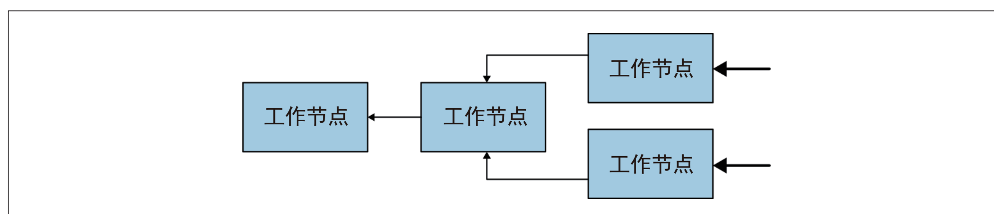


图 4-4：集群中的工作节点相互协作来处理 SQL 语句和数据

4.5 基于连接器的架构

Presto 存储与计算分离的核心是基于连接器的架构。**连接器**为 Presto 提供了连接任意数据源的接口。

每个连接器在底层数据源上提供了一个基于表的抽象。只要数据可以用 Presto 支持的数据类型表示成表、列和行，就可以创建连接器并让查询引擎使用这些数据进行查询处理。

Presto 提供了**服务提供者接口**（service provider interface，SPI）——一种用于实现连接器的 API。通过在连接器中实现 SPI，Presto 就可以在内部使用标准操作来连接到任意数据源并执行操作。连接器负责处理与特定数据源相关的细节。

连接器实现 API 的三个部分：

- 获取表、视图、schema 元数据的操作；
- 产生数据分区的逻辑单位的操作，用于 Presto 的并行读写；
- 在源数据和查询引擎所需的内存数据格式之间进行转换的数据读取和写入组件。

Presto 提供了许多系统的连接器，如 HDFS/Hive、MySQL、PostgreSQL、MS SQL Server、Kafka、Cassandra、Redis 等。第 6 章和第 7 章将介绍其中一些连接器。可用的连接器还在持续增加，你可以从 Presto 文档中获得其支持的连接器的最新列表（参见 1.4.2 节）。

当你需要连接到一个尚未支持的数据源时，也可以利用 Presto 的 SPI 创建自定义的连接器。如果你决定创建新连接器，我们强烈建议你先了解 Presto 的开源社区、寻求我们的帮助并将你的连接器贡献出来，可以参考 1.4 节以获得更多信息。如果组织内有独特的或私有的数据源，可能也需要自定义连接器。这就是 Presto 允许用户使用 SQL 查询任何数据源的原因——真正的 SQL-on-Anything。

图 4-5 展示了 Presto SPI 包含的不同接口，其中协调器使用的是元数据、数据统计和数据定位接口，工作节点使用的是数据流接口。

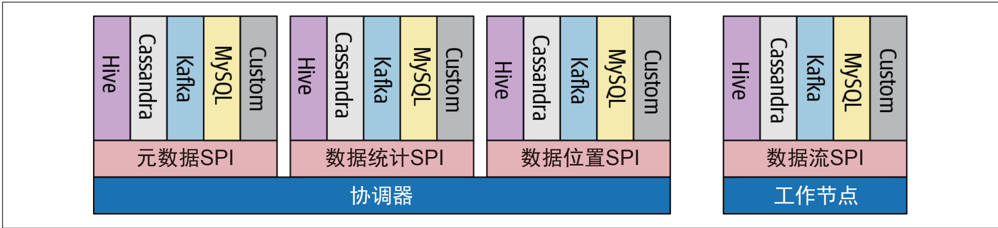


图 4-5：Presto SPI 概览

Presto 服务端程序在启动时以插件的形式加载连接器。连接器由 catalog 属性文件中特定的参数配置，并从插件目录中加载。第 6 章将进一步探索这一问题。



Presto 的很多功能使用了基于插件的架构。除了连接器，插件也可以提供事件监听器、访问控制以及函数和数据类型。

4.6 catalog、schema和表

Presto 集群使用前文所述的基于连接器的架构处理所有查询。每个 catalog 都会配置一个连接器来访问特定的数据源。数据源在 catalog 中暴露出一个或多个 schema，每个 schema 又包含表。表提供数据行，每个数据行由一些具有不同数据类型的列组成。可以在第 8 章（特别是在 8.3 节、8.4 节和 8.6 节）了解 catalog、schema 和表的更多信息。

4.7　查询执行模型

你已经了解，在实际使用中，Presto 通常部署为包含一个协调器和多个工作节点的集群。接下来介绍 Presto 如何处理实际的 SQL 查询语句。



有关 Presto 的 SQL 支持的更多细节，请参见第 8 章和第 9 章。

执行模型是在 Presto 中进行查询性能调优必备的基础知识。

回忆一下，终端用户通过 CLI、使用 ODBC 或 JDBC 的客户端或其他客户端库发送 SQL 语句到协调器，协调器之后调用工作节点从数据源获得数据、创建结果数据集并将结果返回给客户端。

我们先仔细了解一下协调器内部的情况。SQL 语句首先以文本形式提交到协调器，协调器解析和分析这条语句，之后创建一个由 Presto 内部数据结构表示的执行计划，叫作**查询计划**。图 4-6 展示了这一流程。查询计划全面地表示了一条 SQL 语句处理数据和返回结果所需进行的步骤。

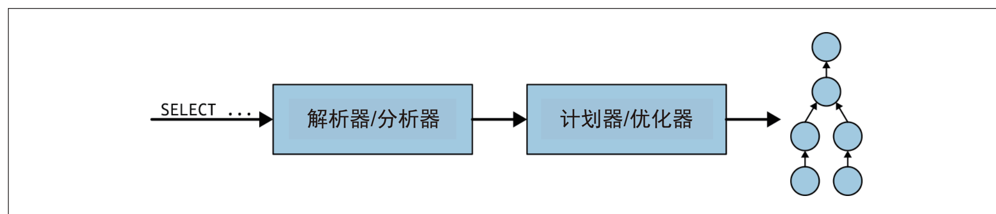


图 4-6：处理一条 SQL 查询语句并创建查询计划

如图 4-7 所示，查询计划生成过程利用了元数据 SPI 和数据统计 SPI 来创建查询计划。也就是说，协调器会使用 SPI 直接连接到数据源，以收集有关表和其他元数据的信息。

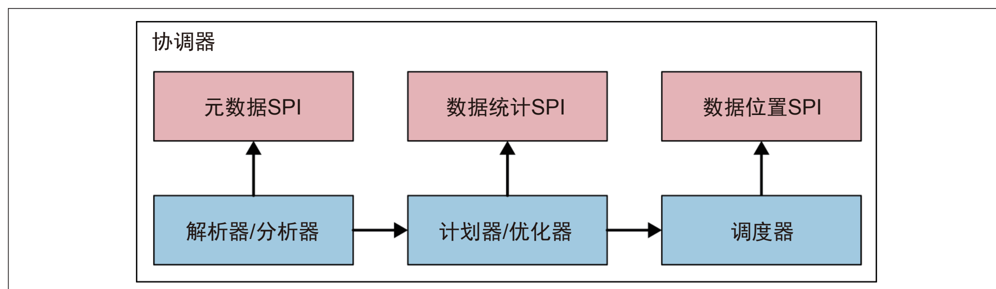


图 4-7：查询计划和调度的 SPI

协调器通过元数据 SPI 获取表、列和数据类型的信息。这些信息用于对查询进行语义校验、类型检查和安全检查。

统计 SPI 用于获取行数和表大小的信息，从而在计划期间进行基于代价的查询优化。

在创建分布式查询计划时会利用数据位置 SPI 来生成表内容的逻辑切片。切片是任务分配和并行的最小单位。



这里对不同类型 SPI 的划分是概念上的，实际的底层 Java API 以更细粒度的形式划分为多个 Java 包。

分布式查询计划是简单查询计划的一个扩展，它包含一个或多个 Stage。简单查询计划被切分为多个**计划片段**（plan fragment）。Stage 是在运行时的计划片段，它包含对应计划片段所描述的所有任务。

协调器将查询计划切分成 Stage，从而分配给集群中的多个工作节点进行并行处理，从而加快整体查询的执行速度。多个 Stage 会被组织成一棵依赖树。Stage 的数量依赖于查询的复杂度。例如，查询的表、返回的列、`JOIN` 语句、`WHERE` 条件、`GROUP BY` 操作和其他 SQL 语句都可能影响 Stage 的数量。

图 4-8 展示了集群中的协调器如何将**逻辑查询计划**转换为**分布式查询计划**。

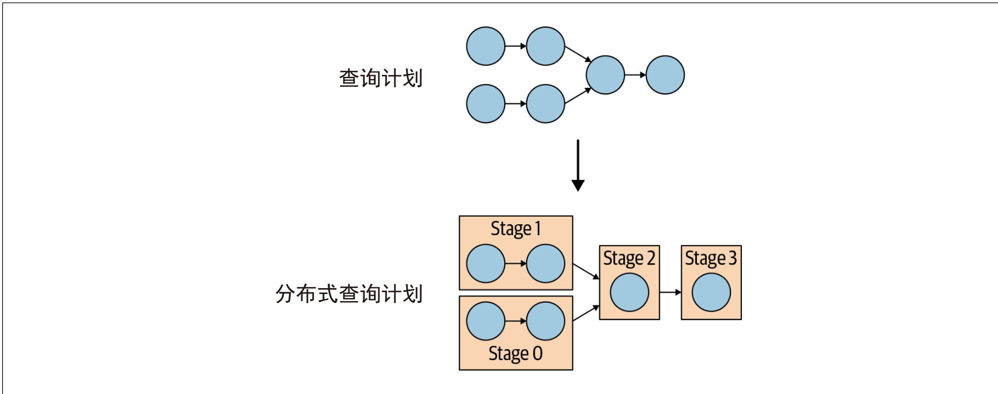


图 4-8：逻辑查询计划到分布式查询计划的转换

分布式执行计划定义了 Stage 和查询在 Presto 集群上执行的方式。协调器使用它在工作节点上进一步计划和调度**任务**。一个 Stage 通常包含一个或多个任务，每个任务则负责处理一小部分数据。

如图 4-9 所示，协调器将一个 Stage 里的任务分配给集群中的工作节点。

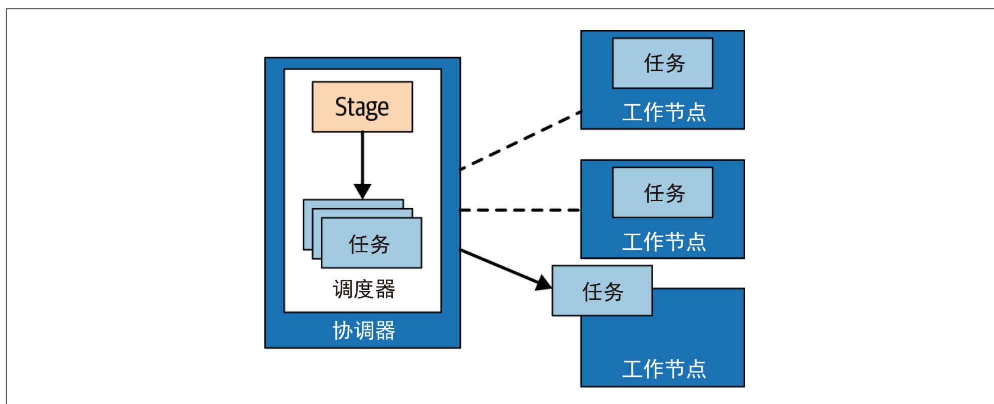


图 4-9：协调器执行的任务管理

一个任务处理数据的单位是**切片**。切片代表一个工作节点可以抽取并处理的一段底层数据，它是并行和任务分配的单位。连接器所执行的特定数据操作取决于底层的数据源。

例如，Hive 连接器用文件的路径、读取偏移量和长度来描述切片信息，这些信息指明了所要处理文件的区域。

源 Stage 的任务以 page[1] 的形式生产数据，每个 page 都是以列式存储格式表示的一系列行。这些 page 传输到下游的中间 Stage。`Exchange` 算子从上游 Stage 中读取数据，从而在不同 Stage 之间传输 page。

在连接器的帮助下，**源任务**使用数据源 SPI 从底层数据源获取数据。这些数据以 page 的形式在 Presto 的查询引擎之中传送。算子根据它们的语义处理接收到的 page 并产生新 page。例如，`Filter` 算子会丢弃过滤掉的行，`Projection` 算子会生成持有新的派生列的 page 等。包含在一个任务里的一串算子叫作**流水线**。流水线中的最后一个算子通常会将它输出的 page 放置在任务的输出缓冲区中。下游任务的 `Exchange` 算子会从上游任务的输出缓冲区中消费 page。所有这些操作都在不同的工作节点上并行运行，如图 4-10 所示。

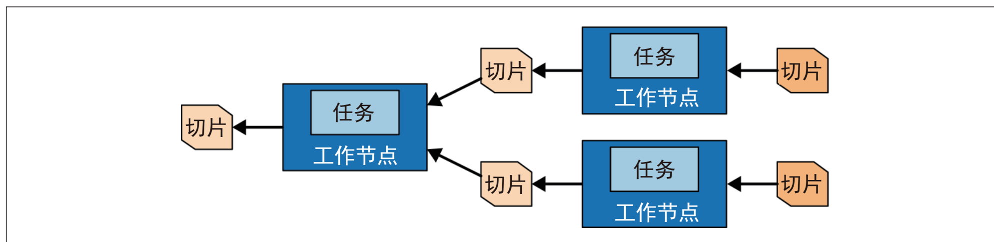


图 4-10：切片中的数据在不同任务之间传输并在不同工作节点上处理

注 1：这里的 page（页面）与一般操作系统语义下的页面不同。——译者注

因此，任务是运行时分配给一个工作节点的计划片段。在任务创建之后，它会为每个切片初始化一个**驱动**。每个驱动都是包含多个算子的流水线的一个实例，并且负责处理切片中的数据。如图 4-11 所示，根据 Presto 配置和环境，一个任务可以使用一个或多个驱动。当所有驱动都执行完且数据被传送到下一个切片时，驱动和任务的工作就结束了，它们之后会被销毁。

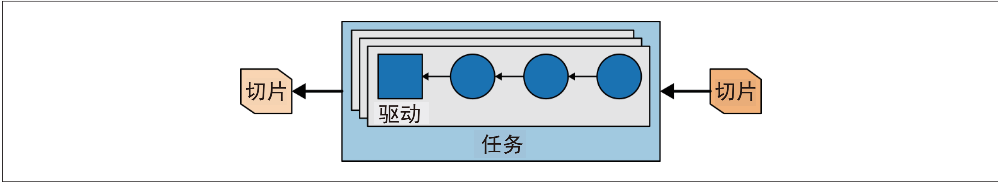


图 4-11：具有输入输出切片的任务及其并行驱动

算子处理输入数据并为下游算子生产输出数据。常见的算子包括 `TableScan`（表扫描）、`Filter`（过滤）、`Join` 和 `Aggregate`（聚合）。一系列相连的算子组成一条算子流水线。例如，你可以拥有一条流水线，它先扫描并读入数据，再过滤数据，最后在数据上执行局部聚合。

要处理一条查询，协调器首先根据来自连接器的元数据创建切片列表。使用该切片列表，协调器开始在工作节点上调度任务，以获取其中的数据。在查询执行期间，协调器跟踪所有可用于处理的切片和任务在工作节点上执行的位置。一些任务完成了处理，并产生了很多供下游处理的切片，协调器就会继续调度更多的任务来处理它们，直到没有待处理的切片为止。

一旦工作节点处理完了所有切片，全部数据就可用了。此时协调器会将结果返回给客户端。

4.8　查询优化

在介绍 Presto 的查询优化器和基于代价的优化之前，我们先来搭建一个舞台，把我们的考虑限定在一定的范围内。我们在一个查询案例的基础上进行探索，以帮助你理解查询优化的过程。

示例 4-1 是一个在 TPC-H 数据集（参见 6.3 节）上的查询，它计算每个国家（地区）订单的累计金额，并列出金额最高的 5 个国家（地区）。

示例 4-1　解释查询优化过程的查询案例

```
SELECT
    (SELECT name FROM region r WHERE regionkey = n.regionkey) AS region_name,
    n.name AS nation_name,
    sum(totalprice) orders_sum
FROM nation n, orders o, customer c
```

```
WHERE n.nationkey = c.nationkey
  AND c.custkey = o.custkey
GROUP BY n.nationkey, regionkey, n.name
ORDER BY orders_sum DESC
LIMIT 5;
```

我们先来理解此 SQL 查询的结构及其目的。

- `SELECT` 语句用于引用 `FROM` 语句中指定的 3 个表（`nation`、`orders` 和 `customer`），并隐式地定义了它们之间的 `CROSS JOIN` 关系。
- `WHERE` 条件语句用于过滤掉 `nation`、`orders` 和 `customer` 3 个表中不匹配的行。
- 聚合语句 `GROUP BY regionkey` 用于将每个国家（地区）的订单价值聚合起来。
- 子查询 (`SELECT name FROM region WHERE regionkey = n.regionkey`) 用于从 region 表中拉取区域名称。注意，这是一个关联子查询，从语义上讲就好像对外层查询的每一行结果都会执行一次此子查询一样。
- 排序语句 `ORDER BY orders_sum DESC` 将结果排序后再返回。
- `LIMIT` 语句将返回结果限制为 5 行，因此只会返回订单金额之和排名前 5 的国家或地区，其他结果会被丢弃。

4.8.1 解析和分析

在对查询的执行进行规划之前，需要先解析和分析查询语句。可以在第 8 章和第 9 章找到有关 SQL 语言及其相关句法规则的细节。Presto 会在解析查询语句时验证其语法，然后分析查询。

识别查询用到的表

表按 catalog 和 schema 进行组织，因此可能会有重名的表。例如，TPC-H 数据集在不同的 schema 下提供不同数据量的 `orders` 表，如 `sf10.orders` 和 `sf100.orders` 等。

识别查询中用到的列

使用列限定引用符 `orders.totalprice` 可以唯一地指向 `orders` 表的 `totalprice` 列。然而，如示例 4-1 所示，一个 SQL 查询通常只使用名称（如 `totalprice`）来引用一个列。Presto 分析器可以决定这样的列来自于哪个表。

识别 ROW 值中的字段引用

解引用表达式 `c.bonus` 既可能指向原名为 `c` 的表中的 `bonus` 列，也可能指向别名为 `c` 的表中的 `bonus` 列。或者，它也可以指向名为 `c`，类型为 `ROW`（一种拥有命名字段的结构体类型）的列中值的 `bonus` 字段。Presto 分析器负责决定哪一种情况是适用的，且如果出现混淆，它会优先选择引用表限定的列。分析的过程需要符合 SQL 语言的作用域和可见性规则。在优化过程中会使用到如限定符消歧等过程收集的信息，因此优化器无须再理解查询语言的作用域规则。

如你所见，查询分析器的职责十分复杂且横跨很多领域。它的角色相当具有技术性，但只要查询语句正确，它对用户就是不可见的。只有当一个查询违反了 SQL 语法，超出了用户权限或出现其他谬误时，分析器才暴露其存在。

当查询分析完成，且所有标识符都被处理和解析后，Presto 就会进入下一个步骤：查询优化。

4.8.2 初始查询计划

查询计划可以被看作产生查询结果的程序。SQL 是一种声明式语言：用户编写 SQL 查询来描述他们想要从系统中获得的数据。用户并不会像在命令式编程中那样指定处理数据和获得结果的步骤。处理数据以获得期望结果的步骤将交由查询优化器决定。

处理数据的一连串步骤通常叫作**查询计划**（query plan）。理论上来说，得到相同查询结果的查询计划的数量可能是指数级的。这些查询计划的性能千差万别，Presto 的查询优化器会试图寻找最优计划。产生相同结果的查询计划叫作**等价计划**。

下面考虑示例 4-1 中的查询语句。此查询最直接的查询计划非常接近其 SQL 语法结构。示例 4-2 展示了这个查询计划。对于我们的讨论来讲，示例 4-1 中的伪代码应该简单易懂。你只需要知道查询计划是树状的，执行从叶子节点开始，沿着树结构逐步上升。

示例 4-2　示例查询计划的手动化简后的直观文本表示

```
- Limit[5]
  - Sort[orders_sum DESC]
    - LateralJoin[2]
      - Aggregate[by nationkey...; orders_sum := sum(totalprice)]
       - Filter[c.nationkey = n.nationkey AND c.custkey = o.custkey]
          - CrossJoin
            - CrossJoin
              - TableScan[nation]
              - TableScan[orders]
            - TableScan[customer]
      - EnforceSingleRow[region_name := r.name]
        - Filter[r.regionkey = n.regionkey]
          - TableScan[region]
```

查询计划中的每个元素都可以用直观的命令式风格实现。例如，`TableScan` 算子从底层存储中访问表，并返回包含表中所有行的结果集。`Filter` 算子接收行并在每一行数据上应用过滤条件，只留下满足条件的行。`CrossJoin` 算子从两个子节点接收数据集，返回两个数据集中行的所有组合，它可能会将其中一个数据集存放在内存中，从而避免多次访问底层存储。



最新版本的 Presto 更改了这些查询计划算子的名称。例如，`TableScan` 与指定了表的 `ScanProject` 是等价的，`Filter` 算子被重命名为 `FilterProject`，但其背后的思想保持不变。

现在我们来考虑下这个查询计划的计算复杂度。在知道实际实现细节之前，我们无法准确地推测一个查询计划的复杂度，但可以假设一个查询计划节点的计算复杂度不会低于其产生数据集的行数。因此，我们可以用描述渐近复杂度下界的大 Omega 表示法来估算复杂度。如果 N、O、C 和 R 分别表示表 `nation`、`orders`、`customer` 和 `region` 表的行数，则有以下内容。

- `TableScan[orders]` 算子读取 `orders` 表并返回 O 行数据，则其复杂度是 $\Omega(O)$。类似地，对 `nation` 和 `customer` 表的 `TableScan` 分别返回 N 和 C 行，因此其复杂度分别为 $\Omega(N)$ 和 $\Omega(C)$。
- `TableScan[nation]` 和 `TableScan[orders]` 之上的 `CrossJoin` 节点返回 `nation` 和 `orders` 表中行的组合（正交积），因此其复杂度是 $\Omega(N \times O)$。
- 再之上的 `CrossJoin` 节点将之前返回 $N \times O$ 行的 `CrossJoin` 节点的输出，与 `TableScan[customer]` 的输出相结合，`customer` 表输出的行数是 C，因此总的复杂度是 $\Omega(N \times O \times C)$。
- 底层的 `TableScan[region]` 复杂度为 $\Omega(R)$，但其上的 `LateralJoin` 会调用它 N 次，聚合之后一共有 N 行被返回，因此总的计算复杂度是 $\Omega(R \times N)$。
- `Sort` 操作需要排序 N 行，它的计算复杂度不小于 $N \times \log(N)$。

我们先暂时忽略其他操作的代价，因为它们与分析过的复杂度相比可忽略不计。之前查询计划的总代价至少是 $\Omega[N + O + C + (N \times O) + (N \times O \times C) + (R \times N) + (N \times \log(N))]$。不知道相对表大小的情况下，此式可以化简为 $\Omega[(N \times O \times C) + (R \times N) + (N \times \log(N))]$。可以合理地假设，`region` 是最小的表而 `nation` 是次小的表，这样我们就可以忽略结果中第二个部分和第三个部分，最终化简的结果是 $\Omega(N \times O \times C)$。

代数公式就到此为止，接下来看一下实际情况。假设一个热门的购物网站有来自 200 个国家（地区）的 1 亿用户，总共产生了 10 亿订单。这两个表的 `CrossJoin` 将产生 2000 亿亿（20 000 000 000 000 000 000）行数据。假如有一个包含 100 个节点的中等大小的集群，每个节点每秒钟可以处理 100 万行数据，则我们需要 63 个世纪才能处理完查询的中间结果。

当然，Presto 不会尝试执行这样原始的计划。但这一初始查询计划连接了 SQL 语法及其语义的世界和查询优化的世界。查询优化的职责就是将初始查询方案转换并演化为一个等价查询方案，在 Presto 集群资源有限的情况下，至少在合理的时间内可以尽可能快地执行该计划。接下来讨论查询优化如何达成这一目标。

4.9 优化规则

本节将介绍 Presto 中实现的一些重要优化规则。

4.9.1 谓词下推

谓词下推（predicate pushdown）可能是最重要和最容易理解的优化规则了。这一规则将过滤条件移动到尽可能接近数据源的位置，因此，数据量在查询开始后尽可能早地开始缩减。在我们的案例中，这一规则将原 `Filter` 算子的一部分条件保留在新的简化 `Filter` 算子中，另一部分和下层的 `CrossJoin` 算子合并为新的 `InnerJoin` 算子。新的查询计划如示例 4-3 所示，为了可读性我们省略了部分内容。

示例 4-3　将 `CrossJoin` 和 `Filter` 转化为一个 `InnerJoin`

```
...
- Aggregate[by nationkey...; orders_sum := sum(totalprice)]
  - Filter[c.nationkey = n.nationkey AND c.custkey = o.custkey]  // 原始Filter算子
    - CrossJoin
      - CrossJoin
        - TableScan[nation]
        - TableScan[orders]
      - TableScan[customer]
...
...
- Aggregate[by nationkey...; orders_sum := sum(totalprice)]
  - Filter[c.nationkey = n.nationkey]              // 简化后的Filter算子
    - InnerJoin[o.custkey = c.custkey]             // 合并后新产生的InnerJoin算子
      - CrossJoin
        - TableScan[nation]
        - TableScan[orders]
      - TableScan[customer]
...
```

原来的 `CrossJoin` 算子被转换为拥有一个等值条件的 `InnerJoin` 算子。简单起见，假设可以在分布式系统里以等于产生行数的计算复杂度实现这一 Join 操作。这意味着，谓词下推规则将一个“至少”具有 $\Omega(N \times O \times C)$ 计算复杂度的 `CrossJoin` 转化为一个“恰好”具有 $\Theta(N \times O)$ 计算复杂度的 Join 操作。

然而，由于 `nation` 表和 `orders` 表之间没有直接的限制条件，谓词下推无法优化这两表之间的 `CrossJoin`。这时 Cross Join 消除规则就开始发挥作用了。

4.9.2 Cross Join消除

在没有基于代价的优化器的情况下，Presto 将 `SELECT` 查询中包含的表按照它们在查询语句中出现的顺序执行 Join 操作。一个重要的例外是产生 Cross Join 时（即表没有 Join 条件）。在绝大多数实际场景下，我们不希望产生 Cross Join。通常执行正交积所产生的行之后会被过滤掉，但执行 Cross Join 本身代价极高，可能永远无法完成。

Cross Join 消除规则对表重新排序，以最小化 Cross Join 的数量（理想状态下可以将其变为 0）。在没有表相对大小的信息时，除非应用 Cross Join 消除规则，否则表的 Join 顺序

不会改变，因此用户可以掌控 Join 顺序。在我们的案例查询上应用 Cross Join 消除规则的效果如示例 4-4 所示。现在两个 Join 都变成了 Inner Join，使 Join 的整体计算代价下降到 $\Theta(C + O) = \Theta(O)$。查询计划的其他部分仍保持原样，因此整体的查询计算代价变为至少 $\Omega[O + (R \times N) + (N \times \log(N))]$。这里，代表 `orders` 表中行数的 O 是计算复杂度的主要组成部分。

示例 4-4 重新排列 Join 以消除 Cross Join

```
...[i]
  - Aggregate[by nationkey...; orders_sum := sum(totalprice)]
    - Filter[c.nationkey = n.nationkey]          // 先过滤nationkey列
      - InnerJoin[o.custkey = c.custkey]         // 然后是Inner Join custkey
        - CrossJoin
          - TableScan[nation]
          - TableScan[orders]
        - TableScan[customer]
...
...
  - Aggregate[by nationkey...; orders_sum := sum(totalprice)]
    - InnerJoin[c.custkey = o.custkey]          // 重新排列为custkey在前面
      - InnerJoin[n.nationkey = c.nationkey]    // nationkey在后面
        - TableScan[nation]
        - TableScan[customer]
      - TableScan[orders]
...
```

4.9.3 TopN

通常，如果一个查询有 `LIMIT` 语句，那么它之前会有一个 `ORDER BY` 语句。没有排序的情况下 SQL 无法保证会返回哪些行作为结果。在我们的查询中也出现了 `ORDER BY` 和 `LIMIT` 的组合。

这样查询的一种执行方式是，先排序所有产生出来的行，然后只保留前面的一部分。这种方法的计算复杂度是 $\Theta($行数 $\times \log($行数$))$，空间复杂度是 $\Theta($行数$)$。但是，排序全部结果而只保留其中一个很小的子集是相当浪费的。因此，有一个优化规则将 `ORDER BY` 和紧跟着的 `LIMIT` 合并成一个 TopN 计划节点。在查询执行时，TopN 节点在堆中维护所需行数的结果，以流处理的方式读取输入数据并更新堆。这将计算复杂度简化为 $\Theta($行数 $\times \log(\mathrm{Limit}))$，空间复杂度简化为 $\Theta(\mathrm{Limit})$。现在查询计算的总代价是 $\Omega[O + (R \times N) + N]$。

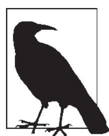
4.9.4 局部聚合

Presto 无须将 `orders` 表中的全部行都发送给 Join，因为不会用到那些独立的订单。我们的案例查询会在每个 nation 上执行累加 `totalprice` 的聚合操作，因此，如示例 4-5 所示，我们可以预聚合这些行。通过聚合数据，我们可以减少流经下游 Join 的数据量。这里返回的结果还不完整，因此称为**预聚合**（pre-aggregation），但产生的数据量可能大大减少，显著地提高了查询性能。

示例 4-5　局部预聚合可以显著提高性能

```
...
  - Aggregate[by nationkey...; orders_sum := sum(totalprice)]
    - InnerJoin[c.custkey = o.custkey]
      - InnerJoin[n.nationkey = c.nationkey]
        - TableScan[nation]
        - TableScan[customer]
      - Aggregate[by custkey; totalprice := sum(totalprice)]
        - TableScan[orders]
...
```

为了提高并行性，这类预聚合采用所谓**局部聚合**（partial aggregation）来实现。这里只显示了简化的查询计划，如果你实际运行 `EXPLAIN` 命令查看查询计划，得到的结果会与这里显示的不同。



示例 4-5 中显示的预聚合并不一定总能提高性能。如果局部聚合不能减少数据量，则其将对查询性能有害。因此，此优化规则目前是默认禁用的，你可以使用 `push_partial_aggregation_through_join` 会话选项来激活它。默认情况下，Presto 将局部聚合放置在 Join 之上以减少 Presto 节点之间通过网络传输的数据量。要充分利用局部聚合，我们可能需要考虑非简化的查询计划。

4.10　实现规则

目前，我们所讨论的都是优化规则，它们的目的是缩短查询处理时间、降低查询内存消耗和减少通过网络传输的数据量。然而，即使在我们的示例查询中，初始查询计划仍包含一个还未被实现的操作：Lateral Join。4.10.1 节介绍 Presto 如何处理这类操作。

4.10.1　Lateral Join去关联化

Lateral Join 可以通过如下方式实现：使用 for 循环迭代一个数据集中的所有行，并对每一行执行另一次的查询。使用这样的实现方法是可能的，但并非 Presto 处理类似场景所采取的方法。相反，Presto 将子查询**去关联化**（decorrelate），它将所有的相关条件拉取上来并形成一个标准的 Left Join。在 SQL 里，这相当于将下列查询：

```
SELECT
    (SELECT name FROM region r WHERE regionkey = n.regionkey)
        AS region_name,
    n.name AS nation_name
FROM nation n
```

变换为

```
SELECT
    r.name AS region_name,
    n.name AS nation_name
FROM nation n LEFT OUTER JOIN region r ON r.regionkey = n.regionkey
```

尽管我们可互换两种查询方式，但熟悉 SQL 语义的细心读者立刻能发现它们并非完全等价。第一个查询在 region 表包含具有相同 regionkey 的重复条目时会执行失败，而第二个查询不会，而是产生更多的结果行。因此，Lateral Join 去关联化在 Join 之外还进行两个额外的操作：首先，它为输入的行编号，使它们可以相互区分；其次，它会在 Join 之后检查是否存在重复行，如示例 4-6 所示。如果检查出重复行，查询处理会以失败告终，因此可以保留原始的查询语义。

示例 4-6　Lateral Join 去关联化需要额外的检查

```
- TopN[5; orders_sum DESC]
  - MarkDistinct & Check
    - LeftJoin[n.regionkey = r.regionkey]
      - AssignUniqueId
        - Aggregate[by nationkey...; orders_sum := sum(totalprice)]
          - ...
     - TableScan[region]
```

4.10.2　Semi-join（IN）去关联化

子查询并不只用于在查询中拉取信息（如上面 Lateral Join 的例子所示），它也常用于配合 IN 谓词过滤行。事实上，IN 谓词既可以用于 Filter（WHERE 语句）也可以用于 Projection（SELECT 语句）。Projection 中的 IN 谓词明显不是 EXISTS 这样一个简单的布尔值运算符。在这里，IN 谓词可以计算为 true、false 和 null。

示例 4-7 展示了一个查询，用于找出客户和物品供应商来自同一个国家（地区）的订单。查询这样的订单非常有用，比如你可以绕过分发中心直接从供应商发货到消费者，以此来降低运送成本或使运送更环保。

示例 4-7　Semi-join（IN）的查询案例

```
SELECT DISTINCT o.orderkey
FROM lineitem l
  JOIN orders o ON o.orderkey = l.orderkey
  JOIN customer c ON o.custkey = c.custkey
WHERE c.nationkey IN (
    -- 多次调用的子查询
    SELECT s.nationkey
    FROM part p
      JOIN partsupp ps ON p.partkey = ps.partkey
      JOIN supplier s ON ps.suppkey = s.suppkey
    WHERE p.partkey = l.partkey
);
```

和 Lateral Join 一样，它可以通过循环迭代外部查询的行并多次调用子查询来实现。这里的子查询会查找一个物品所有的供应商，以及这些供应商所在的所有国家（地区）。

Presto 会通过子查询去关联化的方法来避免这样做——子查询会在去除关联条件的情况下求值一次，之后再用关联条件与外部查询 Join 在一起。这么做最复杂的部分是确保 Join 不

会重复返回结果行（因此使用了去重聚合），以及变换可以正确地保留 `IN` 谓词的逻辑，该逻辑可能返回三个值：`true`、`false` 和 `null`。

在这里，去重聚合使用与 Join 相同的分区方式，因此可以流式执行，无须网络数据交换并且使内存开销降到最低。

4.11 基于代价的优化器

4.8 节讲解了 Presto 的优化器如何将文本查询语句转化为可执行且优化过的查询计划；4.9 节阐释了很多优化规则以及它们对提升查询性能的重要性；4.10 节介绍了使查询计划可以执行所必需的实现规则。

我们从接收用户查询开始，一直介绍到生成最终的执行计划。在这个过程中，我们浏览了一些挑选出来的计划转换方法，这些重要的方法使计划执行的速度有了数量级的提升，甚至让查询的执行变为可能。

现在让我们更深入地了解计划转换，它们在决策时不但基于查询本身的形状，而且更重要的是，它们也将查询数据的形状考虑在内。这就是 Presto 先进的**基于代价的优化器**（cost-based optimizer，CBO）所做的工作。

4.11.1 代价的概念

我们之前使用一个查询案例作为工作模型，为了方便易懂，这次也采用类似的方式。示例 4-8 是之前查询案例的变形，它移除了与本节内容无关的一些查询语句。因此，我们可以专注于查询优化器基于代价的决策。

示例 4-8　基于代价优化的查询案例

```
SELECT
    n.name AS nation_name,
    avg(extendedprice) as avg_price
FROM nation n, orders o, customer c, lineitem l
WHERE n.nationkey = c.nationkey
  AND c.custkey = o.custkey
  AND o.orderkey = l.orderkey
GROUP BY n.nationkey, n.name;
```

如果不基于代价进行决策，查询优化器就会使用规则来优化此查询的初始计划，产生的计划如示例 4-9 所示。这个计划完全由 SQL 查询的语法结构所决定。优化器只使用句法信息进行优化，因此有时被称为**句法优化器**。这个名称故意起得有点滑稽，因为这种优化方法实在非常简单。由于查询计划只基于查询本身，因此你可以通过更改查询中表的句法顺序来手动调整和优化查询。

示例 4-9　来自句法优化器的 Join 顺序

```
- Aggregate[by nationkey...; orders_sum := sum(totalprice)]
  - InnerJoin[o.orderkey = l.orderkey]
   - InnerJoin[c.custkey = o.custkey]
      - InnerJoin[n.nationkey = c.nationkey]
        - TableScan[nation]
        - TableScan[customer]
      - TableScan[orders]
    - TableScan[lineitem]
```

下面假设我们换一种方式来编写这条查询，仅仅改变一下 `WHERE` 语句中条件的顺序：

```
SELECT
    n.name AS nation_name,
    avg(extendedprice) as avg_price
FROM nation n, orders o, customer c, lineitem l
WHERE c.custkey = o.custkey
  AND o.orderkey = l.orderkey
  AND n.nationkey = c.nationkey
GROUP BY n.nationkey, n.name;
```

就产生了一个具有不同 Join 顺序的查询计划：

```
- Aggregate[by nationkey...; orders_sum := sum(totalprice)]
  - InnerJoin[n.nationkey = c.nationkey]
   - InnerJoin[o.orderkey = l.orderkey]
      - InnerJoin[c.custkey = o.custkey]
        - TableScan[customer]
        - TableScan[orders]
      - TableScan[lineitem]
    - TableScan[nation]
```

条件语句的一个简单的改变，就会影响查询计划，进而会影响查询的性能，这对 SQL 分析师来说非常棘手。创建高效的查询也因此需要了解 Presto 处理查询的内部知识。查询的编写者不需要了解如此深入的知识才能充分利用 Presto 的性能。除了人工编写的查询，Apache Superset、Tableau、Qlik 或 MicroStrategy 之类的工具不会生成对 Presto 来说是最优的查询。

CBO 可以确保，这个查询的两个变种可以产生同样的最优查询计划，从而在 Presto 的查询引擎上进行处理。

从时间复杂度的角度来看，无论是将 `nation` 表 Join 到 `customer` 表，还是反过来将 `customer` 表 Join 到 `nation` 表，都无关紧要。两个表都需要处理，在使用 Hash Join 时，总的运行时间与输出行数成正比。然而，时间复杂度并不是唯一重要的事情。通常对于处理数据的程序，尤其是大规模数据库系统来说，内存使用和网络流量也很重要，Presto 也需要将它们考虑在内。要更好地推测 Join 的内存和网络使用，Presto 需要更好地理解 Join 是如何实现的。

无论在单查询还是并发查询任务中，CPU 时间、内存需求和网络带宽的使用都是构成查询执行时间的三个维度，这些维度组成了 Presto 的代价。

4.11.2　Join的代价

使用相等条件（=）Join 两个表时，Presto 可以实现 Hash Join 的扩展版算法。其中一个表称为**构建侧**，这个表的内容以 Join 条件使用的列作为键，构建一个散列查询表。另一个表称为**探测侧**，一旦散列查找表构建完成，就会使用探测侧表的行去构建侧的散列表中以常数时间查找匹配的行。默认情况下，Presto 会使用三层散列以尽可能地并行处理。

1. 基于 Join 条件列的散列值将两个 Join 的表的内容分配到各个工作节点。应该匹配在一起的行具有相同的 Join 条件列，因此也会分配到同一个节点。这样做相当于将问题的规模除以这一 Stage 使用到的节点数。**节点级别的数据分配**是第一层散列。
2. 在节点级别上，构建侧进一步使用散列函数将任务分散到多个工作线程。构建散列表是一个 CPU 密集的过程，使用多线程完成工作可以极大地提高吞吐量。
3. 每个工作线程最后给出完成的散列查询表的一个分区。每个分区本身也是一个散列表。这些分区会合并成第二层的查询散列表，因此我们可以避免将探测侧的处理也分散到多个线程。探测侧依然使用多线程处理，但任务是批量分配到线程上的，这比使用散列函数将探测侧的数据分区处理速度快。

如你所见，构建侧的数据一直存放在内存中，以快速处理数据。当然，此时的内存开销也与构建侧的数据量成正比。这意味着构建侧的数据量不能超过节点上的可用内存，同时也表示可用于其他操作和查询的内存变少了。与 Join 相关的不仅有内存开销，而且还有网络开销。之前介绍的算法会在网络上发送两个 Join 表的内容，以加速节点级别的数据分配。

为了控制 Join 的内存开销，哪个表会成为构建表由 CBO 选择。在特定条件下，优化器可以避免其中一个表的网络传输，这样可以减少网络带宽使用（即减少网络开销）。为了达成这一目的，CBO 需要了解 Join 表的大小，这些数据由表统计信息提供。

4.11.3　表统计信息

4.5 节介绍了连接器的角色，每个表都由连接器提供。除了提供表的 schema 信息和对实际数据的访问，连接器也可以提供表和列的统计信息：

- 表中行的数量；
- 列中不同值的数量（基数）；
- 列中 `NULL` 值所占的比例；
- 列的最大值和最小值；
- 列的平均数据大小。

当然，如果没有某些信息——例如我们不知道一个 VARCHAR 类型的列的平均文本长度——连接器仍可提供其他信息，CBO 可以利用这些信息工作。

有了对 Join 表中行数的估计以及可选列的平均数据长度信息，CBO 就拥有了足够的信息来决定前文中查询案例的最优表排序。它可以从最大的表（lineitem）开始，随后 Join orders 表、customer 表和 nation 表：

```
- Aggregate[by nationkey...; orders_sum := sum(totalprice)]
  - InnerJoin[l.orderkey = o.orderkey]
    - InnerJoin[o.custkey = c.custkey]
      - InnerJoin[c.nationkey = n.nationkey]
        - TableScan[lineitem]
       - TableScan[orders]
      - TableScan[customer]
    - TableScan[nation]
```

这样的计划相当不错并很值得考虑，因为每次 Join 都将具有更小数据量的部分放在构建侧，但这不一定就是最优方案。如果你使用提供表统计信息的连接器并执行上述查询案例，则可以用下面的会话属性来启用 CBO。

```
SET SESSION join_reordering_strategy = 'AUTOMATIC';
```

基于连接器提供的表统计信息，Presto 可能会给出一套不同的计划：

```
- Aggregate[by nationkey...; orders_sum := sum(totalprice)]
  - InnerJoin[l.orderkey = o.orderkey]
    - TableScan[lineitem]
   - InnerJoin[o.custkey = c.custkey]
      - TableScan[orders]
      - InnerJoin[c.nationkey = n.nationkey]
        - TableScan[customer]
        - TableScan[nation]
```

这个计划避免了将最大的表（lineitem）在网络上传输三次，因此被选中成为最终的方案。在此方案中，lineitem 表只会被分发到各个节点一次。

最终的方案取决于 Join 表的实际大小和集群中的节点数，因此当你自己测试这条查询时，你可能会得到与上述计划不同的结果。

细心的读者可能会注意到 Join 顺序的选择只基于 Join 条件、表之间的链接和表的数据量（包括行数和列平均数据长度）。其他的统计信息对于优化更复杂的查询计划很重要，这些查询计划在表查询和 Join 之间包含其他中间操作，如过滤、聚合和非内部 Join（non-inner join）。

4.11.4 过滤统计信息

如前所述，了解查询中涉及的表的大小对于在查询计划中适当地重新排序 Join 表至关重要，但仅仅了解表的大小还不够。考虑之前查询案例的一个变种，用户添加了一个类似

`l.partkey = 638` 的条件，以更深入地在数据集中挖掘某一特定商品的订单信息。

```
SELECT
    n.name AS nation_name,
    avg(extendedprice) as avg_price
FROM nation n, orders o, customer c, lineitem l
WHERE n.nationkey = c.nationkey
  AND c.custkey = o.custkey
  AND o.orderkey = l.orderkey
  AND l.partkey = 638
GROUP BY n.nationkey, n.name;
```

在添加此条件之前，`lineitem` 是最大的表，因此查询优化的重心是此表的处理，但现在过滤后的 `lineitem` 变成 Join 中数据量最小的成员。

下面的查询计划展示了当过滤后的 `lineitem` 表足够小时，CBO 会将其放置在 Join 的构建侧，这样它可以对其他表起到过滤作用。

```
- Aggregate[by nationkey...; orders_sum := sum(totalprice)]
  - InnerJoin[l.orderkey = o.orderkey]
   - InnerJoin[o.custkey = c.custkey]
      - TableScan[customer]
      - InnerJoin[c.nationkey = n.nationkey]
        - TableScan[orders]
        - Filter[partkey = 638]
          - TableScan[lineitem]
    - TableScan[nation]
```

CBO 会使用连接器提供的统计信息来估计 `lineitem` 表过滤后的行数，如列中的不同值的数量和 `NULL` 值所占的比例。对于条件语句 `partkey = 638` 来说，`NULL` 值不满足这个条件，因此优化器可以知道去除 `NULL` 值之后 `partkey` 列所剩的行数。更进一步，如果列中的值大致满足均匀分布，你就可以推算出过滤后的行数：

```
过滤后的行数 = 总行数 * (1 - NULL值比例) / 不相同值的数量
```

显然，上述公式仅在值均匀分布的情况下有效。然而，优化器无须知道过滤后所得的确切行数，而只需要一个估计值。因此，估计值有一定的偏差通常不是问题。当然，如果某个商品（比如 Starburst 牌糖果）的销量远大于其他商品，估计结果就会太偏离实际情况，因此可能优化器会选择一个糟糕的计划。目前，你只能通过禁用 CBO 来处理这种情况。

未来连接器将可以提供有关数据分布的信息，从而处理这种场景。如果有数据分布的直方图，CBO 就可以更精确地估计过滤后的行数。

4.11.5 分区表的统计信息

过滤表的一种特殊情况值得单独讨论：分区表（partitioned table）。使用 Hive 连接器链接的 Hive/HDFS 数据仓库可能将数据组织成**分区表**（参见 6.4 节）。当在分区键上过滤数据

时，查询执行期间只会读取匹配的分区。此外，由于 Hive 中每个分区单独保存表统计信息，因此 CBO 只使用相关分区的统计信息，从而更加精确。

当然，任何连接器都可以为过滤后的关系表提供这类改进后的统计信息，但这里仅参考 Hive 连接器提供这类统计信息的方式。

4.11.6　Join枚举

至此，我们讨论了 CBO 如何使用数据统计信息生成执行查询的最优计划。尤其是，它会选择最优 Join 顺序，该 Join 顺序会显著地影响查询性能，原因如下所示。

Hash Join 的实现

　Hash Join 的实现是非对称的，非常重要的是仔细选择哪一个输入用于构建侧，哪一个用于探测侧。

分布式 Join 的类型

　非常重要的是仔细选择，Join 的输入数据是广播的还是重分布的。

4.11.7　广播Join和分布式Join

在 4.11.6 节中，你了解了 Hash Join 的实现以及选择构建侧和探测侧的重要性。由于 Presto 是分布式系统，因此 Join 可以在一个集群的工作节点上并行执行，每个工作节点处理 Join 的一部分任务。要执行分布式 Join，数据可能需要在网络上分发。根据数据的形状，不同的策略效率可能差别很大。

1. 广播 Join 策略

在**广播 Join 策略**中，Join 的构建侧广播到并行执行 Join 的所有工作节点上。换句话说，每个工作节点上的 Join 都得到构建侧数据的一个完整副本（如图 4-12 所示）。只有当探测侧在工作节点上无重复地分布时，这种策略才会提供正确的语义，否则会产生重复数据。

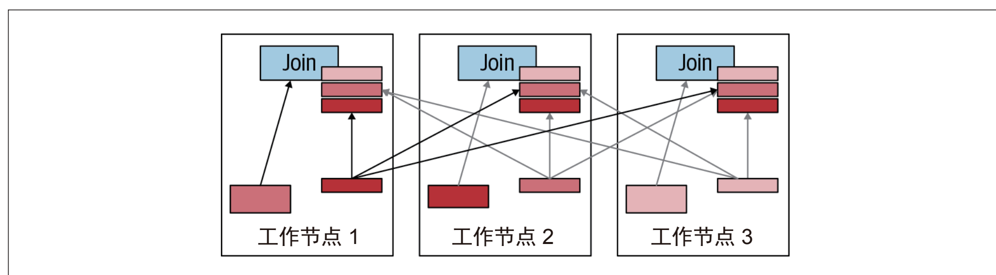


图 4-12：广播 Join 策略的可视化

广播 Join 策略在构建侧数据量很小时非常有优势，它使得数据传输的成本很低。当探测侧的数据量非常大时这一策略也很有优势，因为它避免了像分布式 Join 策略那样重分布探测侧的数据。

2. 分布式 Join 策略

在**分布式 Join 策略**中，构建侧和探测侧的输入数据都需要在集群中重分布，从而使工作节点可以并行执行 Join。与广播 Join 策略不同，分布式 Join 策略下，每个工作节点接收到的都是数据集中独一无二的一部分，而不是数据的相同副本。数据的重分布必须使用分区算法，从而将匹配的 Join 键值发送到同一个节点。例如，在某一特定节点上我们有如下 Join 键的数据集：

```
Probe: {4, 5, 6, 7, 9, 10, 11, 14}
Build: {4, 6, 9, 10, 17}
```

考虑一个简单的分区算法：

```
if joinkey mod 3 == 0 then send to Worker 1
if joinkey mod 3 == 1 then send to Worker 2
if joinkey mod 3 == 2 then send to Worker 3
```

分区后，工作节点 1 上的探测数据和构建数据如下所示：

```
Probe:{6, 9}
Build:{6, 9}
```

工作节点 2 上的探测数据和构建数据如下所示：

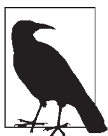
```
Probe: {4, 7, 10}
Build: {4, 10}
```

工作节点 3 上的探测数据和构建数据如下所示：

```
Probe:{5, 11, 14}
Build: {17}
```

通过数据分区，CBO 保证 Join 可以并行处理的同时无须在节点之间共享信息。分布式 Join 的优势在于，Presto 可以计算这样一个 Join：Join 两边的数据量都非常大，而一台机器的内存无法保存构建侧的全部数据。这一策略的缺点是需要额外的网络传输。

广播 Join 策略和分布式 Join 策略之间的选择必须基于代价。两种策略各有取舍，我们必须考虑数据统计信息以获得最优结果。此外，在 Join 重排序过程中也需要选择策略。根据 Join 顺序和 `Filter` 所在位置的不同，数据的形状也会改变。这可能会导致在一种 Join 顺序中分布式 Join 两个数据集更优，而在另一种 Join 顺序中广播 Join 更优，Join 枚举算法已将此考虑在内。



Presto 使用的 Join 枚举算法更加复杂，不在本书讨论范围内，Starburst 的博客文章中对其有详细介绍。这一算法先将问题切分成具有更小分区的子问题，再递归地寻找正确的 Join 使用，最后将分区结果聚合为一个全局结果。

4.12 使用表统计信息

要充分利用 Presto 的 CBO，数据必须带有统计信息。没有统计信息，CBO 能做的非常有限，它需要数据统计信息以估计行数和不同计划的代价。

因为 Presto 自己不存储数据，所以统计信息的提供依赖于连接器的实现。在本书编写的时候，已经有 Hive 连接器可以向 Presto 提供统计信息，如关系数据库连接器在内的其他数据源也可以提供统计信息。例如，PostgreSQL 就可以收集和存储其数据的统计信息，我们可以扩展对应的 PostgreSQL 连接器的实现，以支持将这些信息返回给 Presto 的 CBO。然而，撰写本书时还没有这样的开源连接器。我们期望，经过一段时间的发展，会有更多连接器支持统计信息。你可以持续关注 Presto 的文档以获得这方面的最新信息。

对 Hive 连接器来说，你可以使用以下方法收集统计信息。

- 使用 Presto 的 `ANALYZE` 命令来收集统计信息。
- 启用 Presto 在将数据写入表时收集统计信息的功能。
- 使用 Hive 的 `ANALYZE` 命令来收集统计信息。

重要的是，Presto 将统计信息存储在 Hive Metastore 中，这也是 Hive 存储统计信息的地方。如果你在 Hive 和 Presto 之间共享相同的表，它们会互相覆盖彼此的统计信息。在管理统计信息收集时，应该考虑这一点。

4.12.1 Presto的ANALYZE命令

Presto 提供了 `ANALYZE` 命令用于收集来自某一连接器（如 Hive 连接器）的统计信息。执行时，Presto 会使用执行引擎计算列的统计信息，并将统计信息存放在 Hive Metastore 中。命令的语法如下所示：

```
ANALYZE table_name [ WITH ( property_name = expression [, ...] ) ]
```

例如，要收集和存储 flights 表的统计信息，可以执行如下命令：

```
ANALYZE hive.ontime.flights;
```

在分区的场景下，可以使用 `WITH` 语句来分析特定的分区：

```
ANALYZE hive.ontime.flights WITH (partitions = ARRAY[ARRAY['01-01-2019']])
```

如果你有不止一个分区键，就要用嵌套数组，并将每个键都作为内层数组的元素，最外层的数组用于指定你想要分析的多个分区。在 Presto 中指定分区的能力很有用，例如，你可能有某种类型的 ETL 过程会创建新的分区。当新的数据进入系统时，由于旧的统计信息没有覆盖新的数据，它就过时了。不过，你可以只更新新增分区的统计信息，而不必重新分析所有的数据。

4.12.2 在写入存储时收集数据

对于只通过 Presto 写入数据的表，可以在执行写操作的同时收集统计信息。例如，当你运行 `CREATE TABLE AS` 或 `INSERT SELECT` 查询时，Presto 可以在写入数据到存储（如 HDFS 或 S3）时收集统计信息，然后将其存储在 Hive Metastore 中。

这个功能的有用之处在于免除了手动执行 `ANALYZE` 命令的步骤。这些统计信息永远不会过期。然而，为了让它可以按照预期正常工作，表中的数据必须总是由 Presto 写入。

这一过程的开销已进行了广泛的基准测试，它对性能的影响可忽略不计。要激活这一特性，你需要在使用 Hive 连接器的 catalog 属性文件中添加以下属性：

```
hive.collect-column-statistics-on-write=true
```

4.12.3 Hive的ANALYZE命令

在 Presto 之外，你也可以使用 Hive 的 `ANALYZE` 命令为 Presto 收集统计信息。此时，统计信息的计算由 Hive 而不是 Presto 的执行引擎执行，因此计算结果可能不同。此外，相比自己生成的统计信息，Presto 在使用 Hive 生成的统计信息时总有产生不同行为的风险。通常推荐使用 Presto 自己收集统计信息，但有时可能需要使用 Hive，比如数据由一个更复杂的流水线生成并与其他可能使用到统计信息的工具共享。在 Hive 中收集统计信息，可以执行以下命令：

```
hive> ANALYZE TABLE hive.ontime.flights COMPUTE STATISTICS;
hive> ANALYZE TABLE hive.ontime.flights COMPUTE STATISTICS FOR COLUMNS;
```

有关 Hive 的 `ANALYZE` 命令的完整信息，可以参考 Hive 的官方文档。

4.12.4 显示表统计信息

一旦你收集了统计信息，就常常会想要查看它们。这样可以确认统计信息确实收集好了，也可以在调试性能问题时查看使用了哪些统计信息。

Presto 提供了一个 `SHOW STATS` 命令：

```
SHOW STATS FOR hive.ontime.flights;
```

此外，如果你想要查看一个数据子集的统计信息，就可以给出过滤条件，例如：

```
SHOW STATS FOR (SELECT * FROM hive.ontime.flights WHERE year > 2010);
```

4.13　小结

你现在了解了 Presto 的架构，它使用一个协调器来接收用户请求，之后调用工作节点来组装各种数据。

每个查询都被翻译成分布式查询计划，查询计划由任务组成的多个 Stage 构成。数据以切片形式由连接器返回，并在多个 Stage 中执行，直到最终结果可用并由协调器提供给用户。

如果你对 Presto 架构的更多细节感兴趣，可以研读来自 Presto 创造者的论文“Presto—SQL on Everything”，由 ICDE（IEEE International Conference on Data Engineering）发表。论文也可以在官方网站上找到（参见 1.4.1 节）。

第 5 章会讲解更多有关 Presto 集群部署的知识；第 6 章和第 7 章会介绍使用不同连接器为 Presto 提供更多数据源的信息；第 8 章会阐释如何编写强大的查询。

第 5 章

生产环境部署

你已经在第 2 章中了解了使用 tar.gz 包安装 Presto 的方法，并在第 4 章中了解了 Presto 的架构，现在可以学习安装 Presto 集群的细节了。有了这些知识，你将能够在生产环境中部署具有一个协调器和多个工作节点的 Presto 集群。

5.1 配置细节

Presto 的配置由下文所介绍的多个文件所管理，它们默认存放在安装目录下的 etc 目录中。

配置文件目录以及各个配置文件的默认位置，都可以使用启动器脚本的参数覆盖（参见 5.6 节）。

5.2 服务端配置

Presto 服务端的配置由 etc/config.properties 文件提供。Presto 服务端可以作为协调器或工作节点，也可以兼具两种角色。将服务端程序设定为只执行协调器的工作，并添加一系列其他服务端程序作为专用工作节点，这种配置方式可以提供最佳的性能并创建 Presto **集群**。

上述配置文件的内容非常重要，特别是因为它们决定了服务端程序是作为工作节点还是协调器，而这会影响资源的使用和配置。



Presto 集群中的所有工作节点应该具有完全相同的设置。

接下来介绍 Presto 服务端的一些基础设置属性。在之后的章节中，当讨论认证、授权和资源组等特性时，我们将介绍额外的可选属性。

`coordinator=true|false`
: 设置 Presto 实例是否作为协调器运行，这样可以接受来自客户端的查询请求并管理查询执行。默认值为 `true`，值设为 `false` 会使服务端程序作为工作节点运行。

`node-scheduler.include-coordinator=true|false`
: 设置是否将调度工作运行于协调器上，默认为 `true`。对于更大的集群，我们建议将此属性设为 `false`。在协调器上执行的处理工作会影响查询性能，因为服务端的资源可能被其占用而无法进行服务调度、管理和监控查询执行等重要工作。

`http-server.http.port=8080` 和 `http-server.https.port=8443`
: 指定服务端使用的 HTTP/HTTPS 连接端口。Presto 使用 HTTP 进行内部和外部通信。

`query.max-memory=5GB`
: 查询可以使用的分布式内存的最大值，第 12 章会详细介绍。

`query.max-memory-per-node=1GB`
: 查询在单个机器上可以使用的用户内存的最大值，第 12 章会详细介绍。

`query.max-total-memory-per-node=2GB`
: 查询在任何一台服务器上可以使用的用户内存和系统内存加起来的最大值。系统内存是在执行过程中由读取器、写入器和网络缓冲等组件所使用的内存，第 12 章会详细介绍。

`discovery-server.enabled=true`
: Presto 使用发现服务来找到集群中的所有节点。每个 Presto 实例在启动时都会注册到发现服务。为了简化部署和避免运行额外的服务，Presto 协调器可以运行内置版本的发现服务。它与 Presto 共享 HTTP 服务器，因此使用相同的端口。通常此属性在协调器上设置为 `true`。在所有的工作节点上都必须删除此属性，从而禁用它。

`discovery.uri=http://localhost:8080`
: 发现服务的 URI。使用 Presto 协调器内置的发现服务时，此属性的值应为 Presto 协调器的 URI 且应指定正确的端口。这个 URI 不能以斜杠结尾。

5.3 日志

可选的 Presto 日志配置文件 etc/log.properties 可以让你对命名 Logger 层级设置最低日志级别。每个 Logger 都有名称，通常是使用这个 Logger 的 Java 类的完全限定名。Logger 使用 Java 类的层级。你可以在源代码中看到 Presto 中所有组件使用的包，参见 1.4.4 节。

例如以下日志级别文件：

```
io.prestosql=INFO
io.prestosql.plugin.postgresql=DEBUG
```

第一行将 io.prestosql 包内的所有类（包括嵌套包，如 io.prestosql.spi.connector 和 io.prestosql.plugin.hive）的最低日志级别设置为 INFO。由于默认的日志级别就是 INFO，因此这个例子的第一行设定不会改变任何包的日志级别。将默认级别的设定放在文件中只是为了让配置更加明显。然而，第二行设定将 PostgreSQL 连接器的日志配置覆盖为 DEBUG 级别。

一共有四个日志级别，按照详细程度从高到低排列依次是 DEBUG、INFO、WARN 和 ERROR。本书在讨论 Presto 故障排除等主题时，可能会参考日志设置。



在设置日志级别时，使用 DEBUG 级别可能会输出特别多的内容。请只在实际需要排除故障的特定低级别包上设置 DEBUG 级别，以免产生大量的日志信息，从而对系统性能产生负面影响。

启动 Presto 后，除非你在 etc/node.properties 文件中指定了其他位置，否则将在安装目录下的 var/log 目录中看到多种日志文件。

launcher.log

此文件由启动器（参见 5.6 节）创建，它与服务器的标准输出（stdout）流和标准错误（stderr）流相连，包含服务器初始化时产生的一些日志信息，以及 JVM 产生的错误和诊断信息。

server.log

这是 Presto 主要的日志文件。如果服务器初始化失败，这个文件通常会包含相关的信息。此外，它还包括有关应用程序实际运行和数据源连接等的大部分信息。

http-request.log

HTTP 请求日志，包含服务器接收到的每一个 HTTP 请求。这既包括对 Web UI 和 Presto CLI 的使用请求，也包括第 3 章介绍的 JDBC 或 ODBC 连接请求，因为所有这些操作都使用 HTTP 连接。这个文件也包括认证和授权日志。

所有的日志文件都会自动滚动，你可以在文件大小和是否压缩等细节上进行更详细的配置。

5.4 节点配置

节点配置文件 etc/node.properties 包含了针对某一服务器上安装的单个 Presto 实例（Presto 集群中的一个节点）的配置。

下面是一个简单的示例文件：

```
node.environment=production
node.id=ffffffff-ffff-ffff-ffff-ffffffffffff
node.data-dir=/var/presto/data
```

支持的 Presto 配置属性如下所示。

`node.environment=demo`

运行环境的名称（**必填项**）。Presto 集群里的所有节点必须拥有同一个环境名，该名称显示在 Presto Web UI 的标题部分。

`node.id=some-random-unique-string`

当前 Presto 安装实例的唯一识别符（可选项）。每个节点都必须拥有不同的唯一标识。如果想要在 Presto 节点重启和升级时保持一致的标识，则需要指定该属性。如果不指定此属性，每次重启时都会生成一个新的随机标识。

`node.data-dir=/var/presto/data`

Presto 存储日志文件和其他数据的文件系统目录（可选项），默认为安装目录下的 var 文件夹。

5.5 JVM配置

JVM 配置文件 etc/jvm.config 包含用于启动 Presto 所在 JVM 的命令行选项列表。

这个文件包含一系列的选项，每个选项占据一行。这些选项不是由 shell 解析的，因此包含空格和其他特殊字符的选项无须用引号括起来。

下面的内容是创建 etc/jvm.config 的良好起点：

```
-server
-mx16G
-XX:+UseG1GC
-XX:G1HeapRegionSize=32M
-XX:+UseGCOverheadLimit
-XX:+ExplicitGCInvokesConcurrent
-XX:+HeapDumpOnOutOfMemoryError
-XX:+ExitOnOutOfMemoryError
-Djdk.attach.allowAttachSelf=true
```

因为 `OutOfMemoryError` 通常使得 JVM 进入不一致的状态，所以此时我们会进行堆转储（heap dump），以便调试并强制终止进程。

文件中的 `-mx` 选项是一个重要属性。它设定了 JVM 的最大堆空间，这决定了 Presto 进程能有多少可用内存。

为了使用 Presto，允许 JDK/JVM 吸附（attach）到自身的选项在升级到 Java 11 后是必需的。

关于内存和其他 JVM 设定的更多信息，参见第 12 章。

5.6 启动器

如第 2 章所述，Presto 在 bin 目录下提供了用于启动和管理 Presto 的脚本。这些脚本需要 Python 才能运行。

`run` 命令可用于在前台启动 Presto 进程。

在生产环境，你通常将 Presto 作为一个后台守护进程启动：

```
$ bin/launcher start
Started as 48322
```

上述例子中的数值 `48322` 是进程的 ID（PID），每次启动时都会改变。

你可以使用 `stop` 命令终止一个运行中的 Presto 服务器，这会使它优雅地关闭：

```
$ bin/launcher stop
Stopped 48322
```

当一个 Presto 服务器进程被锁死或遇到其他错误时，可以使用 `kill` 命令强制终止它：

```
$ bin/launcher kill
Killed 48322
```

你可以使用 `status` 命令获取 Presto 的运行状态和 PID：

```
$ bin/launcher status
Running as 48322
```

如果 Presto 进程没在运行，则 `status` 命令返回以下信息：

```
$ bin/launcher status
Not running
```

除了上述命令，启动器脚本还支持许多其他选项，它们可用于自定义配置文件的位置和其他参数。可以使用 `--help` 选项来显示完整的信息：

```
$ bin/launcher --help
Usage: launcher [options] command

Commands: run, start, stop, restart, kill, status

Options:
  -h, --help                show this help message and exit
```

```
  -v, --verbose            Run verbosely
  --etc-dir=DIR            Defaults to INSTALL_PATH/etc
  --launcher-config=FILE   Defaults to INSTALL_PATH/bin/launcher.properties
  --node-config=FILE       Defaults to ETC_DIR/node.properties
  --jvm-config=FILE        Defaults to ETC_DIR/jvm.config
  --config=FILE            Defaults to ETC_DIR/config.properties
  --log-levels-file=FILE   Defaults to ETC_DIR/log.properties
  --data-dir=DIR           Defaults to INSTALL_PATH
  --pid-file=FILE          Defaults to DATA_DIR/var/run/launcher.pid
  --launcher-log-file=FILE Defaults to DATA_DIR/var/log/launcher.log (only in
                           daemon mode)
  --server-log-file=FILE   Defaults to DATA_DIR/var/log/server.log (only in
                           daemon mode)
  -D NAME=VALUE            Set a Java system property
```

其他的安装方法使用这些选项来修改路径。例如，5.8 节中讨论的 RPM 包可以调整路径，从而更好地匹配 Linux 文件系统结构的标准和惯例。你可以使用这些选项满足类似的需求，如服从企业特定的标准、为存储使用特定的挂载点，或仅使用 Presto 安装目录外的路径以便于升级。

5.7 集群安装

第 2 章讨论了如何在单机上安装 Presto，第 4 章介绍了 Presto 是如何设计用于在分布式环境下使用的。

除了用于演示目的，在任何真实使用场景下你都需要将 Presto 安装到一组机器上。幸运的是，集群上的安装和配置与单机上的安装和配置十分相似。你需要将 Presto 安装到每一台机器上，可以手动安装，也可以使用类似 Ansible 这样的自动化部署系统。

目前，你已经将单个 Presto 服务器进程部署为协调器兼工作节点的角色。在集群安装中，你需要安装和配置一个协调器和多个工作节点。

仅需将下载的 tar.gz 归档文件复制到集群中的所有机器上并解压它。

与之前一样，你需要创建 etc 目录并添加相关的配置文件。你可以在本书仓库（参见 1.4.6 节）的 cluster-installation 目录下找到一系列协调器和工作节点的示例配置文件。这些配置文件需要存在于集群的每一台机器上。

这些配置与协调器和工作节点的简单安装模式下的配置基本相同，但有以下重要区别。

- config.properties 文件中的 `coordinator` 属性在协调器上设置为 `true`，在工作节点上设置为 `false`。
- 设置了 `node-scheduler` 属性以排除协调器。
- 在所有的工作节点和协调器上，`discovery-uri` 属性都设置为指向协调器的 IP 地址或主机名。

- 通过删除属性禁用了工作节点上的发现服务。

下面是适用于协调器的主配置文件 etc/config.properties：

```
coordinator=true
node-scheduler.include-coordinator=false
http-server.http.port=8080
query.max-memory=5GB
query.max-memory-per-node=1GB
query.max-total-memory-per-node=2GB
discovery-server.enabled=true
discovery.uri=http://<coordinator-ip-or-host-name>:8080
```

注意适用于工作节点的主配置文件 etc/config.properties 与上述配置的差别：

```
coordinator=false
http-server.http.port=8080
query.max-memory=5GB
query.max-memory-per-node=1GB
query.max-total-memory-per-node=2GB
discovery.uri=http://<coordinator-ip-or-host-name>:8080
```

将 Presto 安装和配置到一组节点上之后，你就可以使用启动器在每一个节点上启动 Presto 了。通常最好先启动 Presto 协调器，再启动工作节点。

```
$ bin/launcher start
```

与之前一样，你可以使用 Presto CLI 来连接到 Presto 服务器。在分布式安装环境下，你需要使用 `--server` 选项指定 Presto 协调器的地址。如果你直接在 Presto 协调器所在的节点上执行 Presto CLI，因为这个选项的默认值是 localhost:8080，所以可以省略它。

```
$ presto --server <coordinator-ip-or-host-name>:8080
```

你现在可以检查 Presto 集群是否正确运行了。nodes 系统表包含集群中所有的活动节点，你可以使用 SQL 查询来查看它们：

```
presto> SELECT * FROM system.runtime.nodes;
 node_id |        http_uri       | node_version | coordinator | state
---------+-----------------------+-------------+---------------------
c00367d  | http://<http_uri>:8080 | 330          | true        | active
9408e07  | http://<http_uri>:8080 | 330          | false       | active
90dfc04  | http://<http_uri>:8080 | 330          | false       | active
(3 rows)
```

返回的列表包含了协调器和所有连接到集群的工作节点。协调器和工作节点使用 REST API 在 /v1/info 路径下暴露其状态和版本信息，例如 http://worker-or-coordinator-host-name/v1/info。

你可以在 Presto Web UI 上确认活动工作节点的数量。

5.8 使用RPM安装

你可以在许多 Linux 发行版上使用 RPM 包管理器安装 Presto，如 CentOS、Red Hat Enterprise Linux 等。

RPM 包托管在 Maven 中央仓库上，地址是 https://repo.maven.apache.org/maven2/io/prestosql/presto-server-rpm。找到所需版本的 RPM 文件并下载它。

可以使用 wget 命令下载对应归档文件，例如，对于版本 330：

```
$ wget https://repo.maven.apache.org/maven2/ \
io/prestosql/presto-server-rpm/330/presto-server-rpm-330.rpm
```

在管理员权限下，你可以使用此归档文件以单节点模式安装 Presto：

```
$ sudo rpm -i presto-server-rpm-*.rpm
```

RPM 安装可以创建基础的 Presto 配置文件和一个服务控制脚本来控制服务器。此脚本配置了 chkconfig，因此会在操作系统启动时自动启动。在使用 RPM 安装了 Presto 之后，你可以使用 service 命令管理 Presto 服务器：

```
service presto [start|stop|restart|status]
```

5.8.1 安装目录结构

使用基于 RPM 的安装方法时，Presto 可以安装到与 Linux 文件系统结构标准更一致的目录结构中。这意味着并非所有文件都包含在之前我们所看到的单一 Presto 安装目录下。对应的服务已经配置过，以便将正确的路径通过启动器脚本传递给 Presto。

/usr/lib/presto/

  此目录包含了运行 Presto 所需的各种库，插件放置在其中的 plugin 目录下。

/etc/presto

  此目录包含了运行 Presto 所需的通用配置文件，如 node.properties、jvm.config 和 config.properties 等。catalog 配置文件位于其中的 catalog 目录下。

/etc/presto/env.sh

  此文件设置了使用的 Java 安装路径。

/var/log/presto

  此目录包含日志文件。

/var/lib/presto/data

  这是数据目录。

/etc/rc.d/init.d/presto

　　这个目录包含了控制服务器进程的服务脚本。

由于我们的目录结构与 Presto 的默认设置不同，node.properties 文件需要下列两个额外的属性：

```
catalog.config-dir=/etc/presto/catalog
plugin.dir=/usr/lib/presto/plugin
```

5.8.2　配置

与 tar.gz 归档文件一样，RPM 包安装 Presto 时充当了便于使用的协调器和工作节点。要创建一个可以工作的集群，你可以手动在集群中节点上修改配置文件，使用 presto-admin 工具，或使用通用配置管理和提供工具，如 Ansible。

5.8.3　卸载Presto

你可以像移除其他任何 RPM 包一样卸载使用 RPM 安装的 Presto：

```
$ rpm -e presto
```

卸载 Presto 时，除日志目录 /var/log/presto 外的所有文件和配置都会被删除。如果你想要保留它们，可以创建一个备份副本。

5.9　在云上安装

Presto 的典型安装会运行一个协调器和多个工作节点。随着时间的推移，集群中工作节点的数量和集群的数量可以随着用户的需求而改变。

所连接数据源的数量和类型及其位置，对于选择在哪里安装和运行 Presto 集群也有重大影响。通常来说，Presto 集群与数据源之间最好能有一个高带宽、低时延的网络连接。

第 2 章所述的运行 Presto 的简单需求，可以让你在多种场景下运行 Presto。你可以在不同类型的机器上运行它，如物理机、虚拟机和 Docker 容器。

Presto 可用于私有云部署，并为许多公有云供应商所用，如 Amazon Web Service（AWS）、Google Cloud Platform（GCP）、Microsoft Azure 等。

容器技术允许你将 Presto 运行在 Kubernetes（k8s）集群上，如 Amazon Elastic Kubernetes Service（Amazon EKS）、Microsoft Azure Kubernetes Service（AKS）、Google Kubernetes Engine（GKE）、Red Hat Open Shift 以及其他任何 Kubernetes 部署。

这类云部署方案的一大优势是提供了部署高弹性集群的可能性，集群中的工作节点可以根据需要创建和销毁。许多用户创建了应用于这类使用场景的工具，包括那些将 Presto 内嵌到产品中的云供应商以及其他提供 Presto 工具和支持的厂商。



Presto 项目本身并没有提供一套完整的、使 Presto 集群便于使用的资源和工具链。使用者通常要创建他们自己的包、配置管理工具、容器镜像、k8s operator 或者其他必要的组件，他们也可能使用类似 Concord 或者 Terraform 这样的工具来创建和管理集群。或者，你也可以选择依赖类似 Starburst 这样的公司所提供的服务和支持。

5.10　集群规模的考量

部署 Presto 的一个重要方面是集群的规模。长期来看，你甚至可能会运行多个针对不同使用场景的集群。调整 Presto 集群的规模是一项复杂的任务，并遵循与其他应用程序一样的模式和步骤。

1. 基于粗略估计和可用的基础设施决定初始规模。
2. 确保集群的工具和基础设施支持集群伸缩。
3. 启动集群并逐步增加使用量。
4. 监控使用率和性能。
5. 根据观察到的结果调整集群的规模和配置。

由监控、调整和继续使用构成的反馈闭环可以让你更好地了解 Presto 部署的行为。

许多因素会影响集群性能，不同 Presto 部署会受这些因素的不同组合的影响。

- 每个节点的资源，如 CPU 和内存。
- 集群内部以及集群到数据源和存储的网络性能。
- 连接的数据源的数量和特性。
- 在数据源上执行的查询及其范围、复杂度、数量和结果数据量。
- 数据源存储的读写性能。
- 活跃用户及其使用模式。

当部署好初始集群后，要确保充分利用 Presto Web UI 进行监控。第 12 章提供了更多信息。

5.11　小结

如你所见，安装和运行 Presto 集群只需少量的配置文件和属性即可。基于实际的基础设施和管理系统，你可以部署一个或多个强大的 Presto 集群。可以在第 13 章查阅真实世界的使用案例。

当然，你还没了解配置 Presto 的一个主要部分——连接到外部数据源，以便用户可以使用 Presto 和 SQL 查询它们。在第 6 章和第 7 章中，你将学习多种数据源和相应的连接器，以及使用连接器访问特定数据源的 catalog 的配置。

第 6 章
连接器

在第 3 章中，你配置了一个 catalog，使用连接器来访问 Presto 中的数据源（尤其是 TPC-H 基准测试数据），然后学习了一些关于如何使用 SQL 查询这些数据的知识。

catalog 是使用 Presto 的一个重要方面。它们定义了与底层数据源和存储系统的连接，并使用了连接器、schema 和表等概念。第 4 章介绍了这些基本概念，第 8 章详细讨论了它们与 SQL 的使用。

连接器将底层数据源（如 RDBMS、对象存储或键值存储）的查询和存储概念翻译成 SQL 和 Presto 中的表、列、行和数据类型的概念。它们可以是简单的 SQL 到 SQL 之间的转换和映射，也可以是更复杂的 SQL 到对象存储或 NoSQL 系统的转换，还可以是由用户定义的。

可以像看待数据库的驱动程序一样看待连接器。它将用户的输入转化为底层数据库可以执行的操作。每个连接器都实现了 Presto **服务提供者接口**（service provider interface，SPI）。这让你能够用同一套 SQL 工具操作任意连接器暴露的底层数据源，从而使 Presto 成为一个 SQL-on-Anything 的系统。

查询性能也受到连接器实现的影响。最基本的连接器与数据源之间建立单条连接，并将数据提供给 Presto。然而，更高级的连接器可以将一条语句分解成多个连接，并行执行操作，从而实现更好的性能。连接器的另一个高级功能是提供表统计信息，它们可以被基于代价的优化器用来创建高性能的查询计划。然而，这样的连接器实现起来会更加复杂。

Presto 提供了众多的连接器。

- 用于 PostgreSQL 或 MySQL 等 RDBMS 的连接器，参见 6.7 节。

- 适用于使用 Hadoop 分布式文件系统（HDFS）和类似对象存储系统查询系统的 Hive 连接器，参见 6.4 节。
- 与非关系数据源的众多连接器，参见 6.5 节。
- 用于 TPC 基准数据的 tpch 和 tpcds 连接器，参见 6.3 节。
- 用于 Java 管理扩展的连接器（即 JMX），参见 6.6 节。

在本章中，你将了解到更多关于 Presto 项目自带的这些连接器的信息。目前，Presto 中已经有二十多个连接器，而且 Presto 团队和用户社区还在创造更多的连接器。商业的、专有的连接器也可用于进一步扩展 Presto 的能力边界和性能。最后，如果你有一个自定义的数据源，或者一个没有连接器的数据源，你可以通过实现必要的 SPI 调用来实现你自己的连接器，然后将它放到 Presto 中的插件目录里中即可。

catalog 和连接器使用的一个重要方面是，所有这些可同时用于 Presto 中的 SQL 语句和查询。这意味着你可以创建跨数据源的查询。例如，你可以将来自关系数据库的数据与存储在对象存储后台的文件中的数据结合起来。这些联邦查询将在 7.6 节中详细讨论。

6.1 配置

如 2.3 节所述，你想访问数据源时需要创建一个 catalog 文件来配置 catalog。在编写查询时，文件的名称决定了 catalog 的名称。

必填属性 `connector.name` 表示 catalog 中使用的是哪个连接器。同一个连接器可以在不同的 catalog 中多次使用。例如，使用相同技术（如 PostgreSQL）访问包含不同数据库的不同 RDBMS 服务器实例。再比如有两个 Hive 集群，你可以在一个 Presto 集群中配置两个 catalog，它们都使用 Hive 连接器，这样你就可以从两个 Hive 集群中查询数据。

6.2 RDBMS连接器示例：PostgreSQL

Presto 包含开源和专有 RDBMS 的连接器，包括 MySQL、PostgreSQL、AWS Redshift 和 Microsoft SQL Server 等。Presto 通过使用每个系统各自的 JDBC 驱动来查询这些数据源。

下面来看一个使用 PostgreSQL 的简单例子。一个 PostgreSQL 实例可能由多个数据库组成，每个数据库都包含多个 schema，而 schema 中则包含表和视图等对象。当在 Presto 中配置 PostgreSQL 时，你需要将目标数据库设置为 catalog。

创建一个简单的 catalog 文件，该文件指向服务器中的特定数据库，etc/catalog/postgresql.properties 文件如下所示，重启 Presto 后，就可以查询到更多信息了。你还可以看到 `postgresql` 连接器已经配置为 `connector.name` 的值了：

```
connector.name=postgresql
connection-url=jdbc:postgresql://db.example.com:5432/database
connection-user=root
connection-password=secret
```



catalog 属性文件中的用户名和密码决定了对底层数据源的访问权限。例如，这可以用来限制只能进行只读操作或限制可以访问哪些表。

可以列出所有的 catalog 以确认新的 catalog 是否可用，并通过 Presto CLI 或使用 JDBC 驱动的数据库管理工具来查看其详细信息（3.1 节和 3.2 节中有详细的解释）：

```
SHOW CATALOGS;
  Catalog
------------
 system
 postgresql
(2 rows)

SHOW SCHEMAS IN postgresql;
  Catalog
------------
 public
 airline
(2 rows)

USE postgresql.airline
SHOW TABLES;
  Table
---------
 airport
 carrier
(2 rows)
```

在上述例子中我们连接到一个 PostgreSQL 数据库，其中包含两个 schema：`public` 和 `airline`。而在 `airline` schema 中包含两个表：`airport` 和 `carrier`。下面尝试运行一个查询，在这个例子中，我们向 Presto 发出一个 SQL 查询，该目标表存在于 PostgreSQL 数据库中。使用 PostgreSQL 连接器，Presto 能够获取要处理的数据，随后将结果返回给用户：

```
SELECT code, name FROM airport WHERE code = 'ORD';
 code |             name
------+------------------------------
 ORD  | Chicago O'Hare International
(1 row)
```

如图 6-1 所示，客户端将查询提交给 Presto 协调器。它将工作分配给一个工作节点，由工作节点使用 JDBC 将整个 SQL 查询语句发送到 PostgreSQL。PostgreSQL JDBC 驱动包含于 PostgreSQL 连接器，PostgreSQL 处理查询并通过 JDBC 返回结果。连接器读取结果并将其

写入 Presto 内部数据格式。Presto 持续在工作节点上进行处理，将其处理结果提供给协调器，然后将最终结果返回给用户。

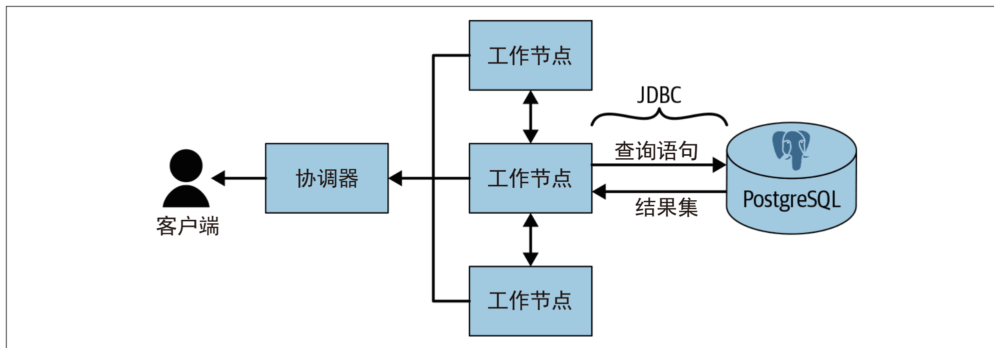


图 6-1：使用连接器中的 JDBC 实现 Presto 集群与 PostgresSQL 的交互

6.2.1 查询下推

可以在前面的例子中看到，Presto 能够通过将 SQL 语句下推到底层数据源中来进行负载的下推处理，这就是所谓的**查询下推**，也称为 **SQL 下推**。这是有利的，因为可以减少底层系统返回给 Presto 的数据量，从而避免了不必要的内存、CPU 和网络开销。此外，像 PostgreSQL 这样的系统通常在某些过滤列上有索引，这样可以更快地处理数据。但是，不一定总是能将整个 SQL 语句下推到数据源中。目前，Presto 连接器的 SPI 限制只能将过滤和列投影下推：

```
SELECT state, count(*)
FROM airport
WHERE country = 'US'
GROUP BY state;
```

对于上述 Presto 查询，PostgreSQL 连接器将构造如下 SQL 查询，从而下推到 PostgreSQL：

```
SELECT state
FROM airport
WHERE country = 'US';
```

当查询被 RDBMS 连接器下推时，有两个重要的地方需要注意。`SELECT` 列表中的列被设置为 Presto 所需要的具体内容。在本例中，我们只需要 `state` 列来处理 Presto 中的 `GROUP BY`。我们还下推了 `Filter country = 'US'`，这意味着无须在 Presto 中对 `country` 这一列进行进一步的处理。可以注意到，聚合没有下推到 PostgreSQL 中，这是因为 Presto 无法下推任何其他形式的查询，因此聚合必须在 Presto 中执行。这可能是有益的，因为 Presto 是一个分布式查询处理引擎，而 PostgreSQL 不是。

如果想将额外的处理下推到底层 RDBMS 源，可以使用视图来实现。如果你在 PostgreSQL

中把处理封装在视图中，它会以表的形式暴露在 Presto 中，对它的处理就会发生在 PostgreSQL 中。假设你在 PostgreSQL 中创建了视图：

```
CREATE view airline.airports_per_us_state AS
SELECT state, count(*) AS count_star
FROM airline.airport
WHERE country = 'US'
GROUP BY state;
```

当你在 Presto 中运行 `SHOW TABLES` 时，就会看到这个视图：

```
SHOW TABLES IN postgresql.airline;
  Table
---------
 airport
 carrier
 airports_per_us_state
(3 rows)
```

现在你可以直接查询视图，所有的处理都被下推到 PostgreSQL，因为视图是以普通表的形式出现在 Presto 中的。

```
SELECT * FROM airports_per_us_state;
```

6.2.2 并行性和并发性

目前，所有的 RDBMS 连接器都使用 JDBC 与底层数据源进行单一连接。即使底层数据源是一个并行系统，也不能并行读取数据。对于像 Teradata 或 Vertica 这样的并行系统，你必须编写并行连接器，以利用这些系统分布式存储数据的特性。

当从同一个 RDBMS 中访问多个表时，对查询中的每个表都会创建一个 JDBC 连接。如果查询在 PostgreSQL 中的两个表之间执行 Join，Presto 通过 JDBC 创建两个不同的连接来检索数据，如图 6-2 所示。它们并行运行，返回结果，然后在 Presto 中执行连接。

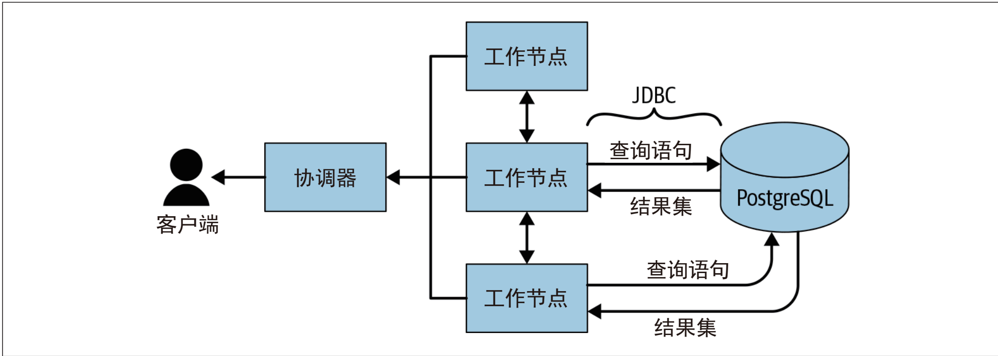


图 6-2：用于访问 PostgreSQL 中不同表的多个 JDBC 连接

就像聚合一样，Join 不能下推。但是，如果你想利用底层 PostgreSQL 系统中可能的性能提升，可以在 PostgreSQL 中创建一个视图，甚至可以添加原生索引来进一步提高性能。

6.2.3 其他RDBMS连接器

目前，Presto 开源项目有四个 RDBMS 连接器。MySQL、PostgreSQL、AWS Redshift 和 Microsoft SQL Server 的连接器已包含在 Presto 的插件目录中，可以随时配置。如果你有多个服务器，或者想要分开访问，你可以在 Presto 中为每个实例配置多个 catalog。只需将 *.properties 文件换个名称即可。与之前一样，属性文件的名称决定了 catalog 的名称：

```
SHOW CATALOGS;
  Catalog
------------
 system
 mysql-dev
 mysql-prod
 mysql-site
(2 rows)
```

不同的 RDBMS 之间存在着微小的差异，下面来看看其各自的 catalog 配置文件是如何配置的。

在 MySQL 中，数据库和 schema 没有区别，catalog 文件和 JDBC 连接字符串都指向具体的 MySQL 服务器实例：

```
connector.name=mysql
connection-url=jdbc:mysql://example.net:3306
connection-user=root
connection-password=secret
```

PostgreSQL 做了明确的区分，一个实例可以有多个包含 schema 的数据库。JDBC 连接指向某个特定的数据库。

```
connector.name=postgresql
connection-url=jdbc:postgresql://example.net:5432/database
connection-user=root
connection-password=secret
```

AWS Redshift 的 catalog 看起来与 PostgreSQL 类似。事实上，Redshift 使用的是 PostgreSQL 的 JDBC 驱动，因为它是基于开源的 PostgreSQL 代码，而且 JDBC 驱动是兼容的，所以可用于：

```
connector.name=redshift
connection-url=jdbc:postgresql://example.net:5439/database
connection-user=root
connection-password=secret
```

Microsoft SQL Server 的连接字符串看起来与 MySQL 类似。但是，SQL Server 确实有数据库和 schema 的概念，这个例子只是简单地连接到默认的数据库：

```
connector.name=sqlserver
connection-url=jdbc:sqlserver://example.net:1433
connection-user=root
connection-password=secret
```

而使用如 `sales` 等不同的数据库时，则必须要配置一个属性。

```
connection-url=jdbc:sqlserver://example.net:1433;databaseName=sales
```

6.2.4 安全性

目前，RDBMS 连接器进行认证的唯一方式是在 catalog 配置文件中存储用户名和密码。由于 Presto 集群中的机器被设计成可信任的系统，这对于大多数的使用来说应该是足够的。为了保证 Presto 和所连接的数据源的安全，对机器和配置文件的安全访问非常重要，应该像对待私钥一样对待它。所有的 Presto 用户都使用相同的连接访问 RDBMS。

如果你不想用明文存储密码，有一些方法可以通过 Presto 客户端传递用户名和密码。第 10 章将进一步讨论。

总而言之，Presto 与 RDBMS 一起使用是非常容易的，可以将所有的系统都暴露在一个地方，并且可以同时查询。仅此就表明使用 Presto 提供了巨大的益处。当然，用其他连接器添加更多的数据源之后，事情就会变得更加有趣。所以，让我们继续深入。

6.3 Presto TPC-H和TPC-DS连接器

第 2 章已经介绍了 TPC-H 连接器的使用方法，下面来详细学习一下。

TPC-H 和 TPC-DS 连接器内置于 Presto 中，并提供了一套 schema 以支持 TPC-H 和 TPC-DS 基准测试。事务处理性能委员会提供的数据库基准测试套件是数据库系统的行业标准，用于衡量高度复杂的决策支持数据库的性能。

这些连接器可用于测试 Presto 的能力和查询语法，无须配置外部数据源的访问。当你查询一个 TPC-H 或 TPC-DS 模式时，连接器通过使用一个确定性算法实时生成一些数据。

创建一个 catalog 属性文件 etc/catalog/tpch.properties，用于配置 TPC-H 连接器：

```
connector.name=tpch
```

TPC-DS 连接器的配置类似。例如，用 etc/catalog/tpcds.properties：

```
connector.name=tpcds
```

这两个连接器所展示的 schema 在结构上包括了相同的数据集：

```
SHOW SCHEMAS FROM tpch;
       Schema
--------------------
 information_schema
 sf1
 sf100
 sf1000
 sf10000
 sf100000
 sf300
 sf3000
 sf30000
 tiny
(10 rows)
```

表 6-1 显示了不同的 schema 如何在 orders 等事务性表中包含越来越多的记录。

表6-1：不同TPCH schema中订单表的记录数示例

schema	count
tiny	15 000
sf1	1 500 000
sf3	4 500 000
sf100	150 000 000

如第 8 章和第 9 章所述，你可以使用这些数据集来学习更多 Presto 所支持的 SQL，而无须连接另一个数据库。

这些连接器的另一个重要使用场景是获得简单可用的数据。可以将这些连接器用于开发和测试，甚至在 Presto 的生产部署中使用。由于易于获得很大的数据量，因此你可以构建查询并在 Presto 集群上施加很大负载。这可以使你更好地了解集群性能，调整和优化它，并确保即使发生了跨版本更新或其他变更，它也可以正常工作。

6.4　用于分布式存储数据源的Hive连接器

如 1.5 节所述，Presto 是为了在 Facebook 这样的体量上运行快速查询而设计的。鉴于 Facebook 在其 Hive 数据仓库中拥有海量的存储，Hive 连接器自然成为 Presto 最早开发的连接器之一。

6.4.1　Apache Hadoop和Hive

在阅读 Hive 连接器以及它对众多对象存储格式的适用性之前，需要先梳理一下 Apache Hadoop 和 Apache Hive 的相关知识。

如果你对 Hadoop 和 Hive 不熟悉且想了解更多，推荐你浏览一下相关项目的官方网站、视频和网络上的其他资源，以及一些优秀的图书。比如 Edward Capriolo 等人的 *Programming Hive*（O'Reilly 出版）就是一本很好的指南。

下面我们先讨论某些 Hadoop 和 Hive 的概念，以便为 Presto 的使用提供足够的背景知识。

Hadoop 的核心是由 HDFS 和应用软件（如 Hadoop MapReduce）组成，这些应用程序与 HDFS 中的数据进行交互。Apache YARN 用于管理 Hadoop 应用程序所需的资源。Hadoop 是一种领先的多机集群系统，它用于大规模数据集的分布式处理。它能够对系统进行伸缩，同时在计算机集群之上提供高可用服务。

最初，数据处理是通过编写 MapReduce 程序来完成的。开发者遵循一种特定的编程模型，以让数据处理能够自然地分布在集群中。这种模型运行得很好，也很稳健。然而，编写 MapReduce 程序来分析问题很麻烦。对于依赖 SQL 和数据仓库的现有基础设施、工具和用户来说，很难迁移到 MapReduce 上。

Hive 提供了 MapReduce 之外的另一种使用方式。它最初是用来在 Hadoop 之上提供一个 SQL 抽象层，从而使用类似于 SQL 的语法与 HDFS 中的数据进行交互。这样，大量了解 SQL 的用户便可以与存储在 HDFS 中的数据进行交互。

Hive 数据以文件的形式存储在 HDFS 中，通常叫作**对象**。这些文件使用各种格式，如 ORC、Parquet 等。这些文件以 Hive 所设定的特定目录和文件布局来存储，如分区表和分桶表。我们把这种布局称为 Hive 风格的表格式。

Hive 的元数据描述了存储在 HDFS 中的数据如何映射到 schema、表和列中，并通过 SQL 进行查询。这些元数据信息保存在 MySQL 或 PostgreSQL 等数据库中，可以通过 **Hive Metastore 服务**（HMS）进行访问。

Hive 运行时提供了类似 SQL 的查询语言和分布式执行层来执行查询。Hive 运行时将查询翻译成一组可以在 Hadoop 集群上运行的 MapReduce 程序。随着时间的推移，Hive 的演进使得查询可以翻译到其他的执行引擎，如 Apache Tez 和 Spark 等。

Hadoop 和 Hive 在业界得到了广泛的应用。随着它们的使用，HDFS 格式已经成为许多其他分布式存储系统所支持的格式，如 Amazon S3 和 S3 兼容的存储、Azure Data Lake Storage、Azure Blob Storage 以及 Google Cloud Storage 等。

6.4.2 Hive连接器

Presto 的 Hive 连接器允许你连接到 HDFS 对象存储集群。它利用 HMS 中的元数据，查询和处理存储在 HDFS 中的数据。

Presto 最常见的使用场景可能是利用 Hive 连接器从分布式存储（如 HDFS 或云存储）读取数据。



Presto 和 Presto Hive 连接器完全不使用 Hive 运行时环境。Presto 是它的替代品，用于运行交互式查询。

Hive 连接器允许 Presto 从 HDFS 中读写，但它并不局限于 HDFS，而是旨在可以与一般的分布式存储一起工作。目前，可以将 Hive 连接器配置为与 HDFS、AWS S3、Azure Blob Storage、Azure Data Lake Storage、Azure Data Lake Storage、Google Cloud Storage 和 S3 兼容存储一起工作。S3 兼容存储可能包括 MinIO、Ceph、IBM Cloud Object Storage、SwiftStack、Cloudian、Riak CS、LeoFS、OpenIO 等。市面上有各种 S3 兼容存储，只有它们实现 S3 API 并且行为方式相同，那么在大多数情况下 Presto 就无须知道它们的区别。

由于 Hadoop 和其他 HDFS 兼容系统的广泛使用，以及 Hive 连接器内置的支持这些系统的扩展功能集，Hive 连接器可以被认为是使用 Presto 查询对象存储的主要连接器。因此，对许多甚至大多数 Presto 用户来说，它至关重要。

在架构上，Hive 连接器与 RBDMS 和其他连接器有一点不同。因为它根本不使用 Hive 引擎本身，所以不能将 SQL 处理下推到 Hive 中。相反，Hive 连接器只是使用 HMS 中的元数据，并使用 Hadoop 项目提供的 HDFS 客户端直接访问 HDFS 上的数据。它还假定数据以 Hive 表格式保存在分布式存储中。

无论使用什么存储系统，schema 信息都是从 HMS 中获取的，并且数据布局也与 Hive 数据仓库相同。概念是一样的，数据却可能并不存储在 HDFS 上。然而，与 Hadoop 不同的是，这些非 Hadoop 的分布式文件系统并非总有 HMS 等价物来存储元数据，以供 Presto 使用。要利用 Hive 风格的表格式，必须将 Presto 配置为要么使用现有 Hadoop 集群中的 HMS，要么使用独立的 HMS。这意味着，你需要使用 AWS Glue 等 HMS 的替代品，或者仅运行 HMS 的一个最小 Hadoop 部署。

使用 HMS 来描述 HDFS 以外的 blob 存储中的数据，使得 Hive 连接器可以查询这些存储系统。这样，通过 Presto 和任何能够使用 SQL 的工具，我们便能够利用 SQL 的所有功能来分析存储在这些系统中的数据了。

配置一个 catalog 来使用 Hive 连接器，需要你创建一个 catalog 属性文件，文件名为 etc/catalog/s3.properties、etc/catalog/gcs.properties 或 etc/catalog/minio.properties，甚至仅仅是 etc/catalog/hdfs.properties 或 etc/catalog/objectstorage.properties。下面假设使用 etc/catalog/hive.properties。你至少需要配置连接器的名称和 HMS 的 URL。

```
connector.name=hive-hadoop2
hive.metastore.uri=thrift://example.net:9083
```

许多其他的配置属性适用于不同的使用场景，其中一些你很快就会了解到。当有疑问时，一定要查看文档（参见 1.4.2 节）。接下来看其中的一些细节。

6.4.3 Hive式表格式

一旦配置好了连接器，就可以在 Presto 中创建 schema，例如，在 HDFS 中创建 schema：

```
CREATE SCHEMA hive.web
WITH (location = 'hdfs://starburst-oreilly/web')
```

schema，有时仍称为数据库，可以包含多个表。你可以在 6.4.4 节中读到更多关于它们的内容。schema 的创建通常只是在 HMS 中创建关于 schema 的元数据：

```
...
hdfs://starburst-oreilly/web/customers
hdfs://starburst-oreilly/web/clicks
hdfs://starburst-oreilly/web/sessions
...
```

使用 Amazon S3 没有太大区别。你只是使用了不同的协议字符串：

```
CREATE SCHEMA hive.web
WITH (location = 's3://example-org/web')
...
s3://example-org/web/customers
s3://example-org/web/clicks
s3://example-org/web/sessions
...
```

6.4.4 内部表与外部表

介绍完 schema，我们需要了解 schema 中的内容，它们是以表的形式组织起来的。Hive 区分了内部表和外部表。**内部表**是由 Hive 管理的，因此其实也是由 Presto 管理的，它和它的数据一起创建于数据库目录所在的位置；而**外部表**不是由 Hive 管理的，它明确指向 Hive 所管理的目录之外的一个位置。

内部表和外部表的主要区别在于，Hive 以及 Presto 是否掌握内部表中的数据。如果你删除一个内部表，HMS 中的元数据和数据本身都会被删除。如果你删除一个外部表，数据仍然存在，只有关于该表的元数据会被删除。

你使用的表的类型取决于你打算使用 Presto 的方式。你可能会将 Presto 用于数据联邦、数据仓库或数据湖，或两者兼而有之，或其他混用方式。你需要决定谁掌握这些数据。可以是 Presto 与 HMS 一起工作，也可以是 Hadoop 和 HMS，或 Spark，或 ETL 管道中的其他工具。在所有情况下，元数据都是在 HMS 中管理的。

哪个系统掌握并管理 HMS 和数据这一决定通常取决于数据架构。在 Presto 的早期使用场

景中，Hadoop 往往控制着数据的生命周期，但随着更多的用例将 Presto 作为中心工具，很多用户的使用模式发生了转变，Presto 接管了控制权。

一些 Presto 的新用户一开始是查询现有的 Hadoop 部署。这种情况下，Presto 一开始更多是作为数据联邦使用，而 Hadoop 掌握数据。你配置 Hive 连接器，将 Hadoop 中的现有表暴露给 Presto 进行查询，此时使用的是外部表，通常不允许 Presto 写入这些表。

其他的 Presto 用户可能从 Hadoop 完全迁移到 Presto，或者开始使用其他对象存储系统，具体来说，往往是基于云的系统。在这种情况下，最好是在 Presto 上执行数据定义语言（DDL），让 Presto 掌握数据。

```
CREATE TABLE hive.web.page_views (
  view_time timestamp,
  user_id bigint,
  page_url varchar,
  view_date date,
  country varchar
)
```

在这个例子中，表 `page_views` 在一个名为 page_views 的目录下存储数据。这个 page_views 目录是由 hive.metastore.warthere.dir 定义的目录下的一个子目录，如果你在创建 schema 时定义了其位置，那么该表的位置就是所指定 schema 目录下的一个子目录。

下面是一个 HDFS 的例子。

```
hdfs://user/hive/warehouse/web/page_views/...
```

下面是一个 Amazon S3 的例子。

```
s3://example-org/web/page_views/...
```

接下来看一下在指向现存数据的 Presto 表上的 DDL。这些数据是通过其他方式创建和管理的，例如由 Spark 或 ETL 过程将数据写入存储中。这种情况下，可以通过 Presto 创建一个外部表，指向这个数据所在的外部位置。

```
CREATE TABLE hive.web.page_views (
  view_time timestamp,
  user_id bigint,
  page_url varchar,
  view_date date,
  country varchar
)
WITH (
  external_location = 's3://starburst-external/page_views'
)
```

这将有关表的元数据插入 HMS 中，包括外部路径和标志，告诉 Presto 和 HMS 该表是外部的，因此由另一个系统管理。

因此，位于 s3://example-org/page_views 中的数据可能已经存在。一旦在 Presto 中创建了表，你就可以开始查询它了。当你将 Hive 连接器配置到现有的 Hive 仓库中时，可以看到现有的表，并且能够立即对这些表进行查询。

或者，你可以在空的目录中创建表，并期望数据被 Presto 或外部源加载进来。在这两种情况下，Presto 都要求已经创建了目录结构；否则，DDL 会出错。创建外部表最常见的场景是与其他工具共享数据时。

6.4.5 分区数据

目前，你已经了解了一个表的数据，不管是内部的还是外部的，都是以一个或多个文件的形式存储在一个目录中。**数据分区**是这一点的延伸，它将逻辑表横向划分为小块数据，称为分区。

这个概念本身源于 RDBMS 中的分区 schema。Hive 将这种技术引入 HDFS 中的数据，用于实现更好的查询性能并提升数据的可管理性。

在分布式文件系统（如 HDFS）和对象存储（如 S3）中，分区已成为标准的数据组织策略。

让我们用这个表的例子来演示一下分区：

```
CREATE TABLE hive.web.page_views (
  view_time timestamp,
  user_id bigint,
  page_url varchar,
  view_date date
)
WITH (
  partitioned_by = ARRAY['view_date']
)
```



`partitioned_by` 子句中列出的列必须是 DDL 中定义的最后一列，否则，Presto 会报错。

与非分区表一样，`page_views` 表的数据位于 .../page_views 中。使用分区可以改变表的布局构建方式。使用分区表在表目录中添加了额外的子目录。在下面的例子中，可以看到由分区键定义的目录结构：

```
...
.../page_views/view_date=2019-01-14/...
.../page_views/view_date=2019-01-15/...
.../page_views/view_date=2019-01-16/...
...
```

Presto 同样使用这种 Hive 风格的表格式。此外，你可以选择在多个列上进行分区：

```
CREATE TABLE hive.web.page_views (
  view_time timestamp,
  user_id bigint,
  page_url varchar,
  view_date date,
  country varchar
)
WITH (
  partitioned_by = ARRAY['view_date', 'country']
)
```

当选择多个分区列时，Presto 会创建一个分层目录结构：

```
...
.../page_views/view_date=2019-01-15/country=US...
.../page_views/view_date=2019-01-15/country=PL...
.../page_views/view_date=2019-01-15/country=UA...
.../page_views/view_date=2019-01-16/country=US...
.../page_views/view_date=2019-01-17/country=AR...
...
```

分区可以提高查询性能，特别是当你的数据规模越来越大时。让我们以下面的查询为例：

```
SELECT DISTINCT user_id
FROM page_views
WHERE view_date = DATE '2019-01-15' AND country = 'US';
```

当提交此查询时，Presto 会识别 `WHERE` 子句中的分区列，并使用关联值只读取 view_date=2019-01-15/country=US 子目录。只读取需要的分区可能会节省大量性能。如果数据规模现在还很小，性能提升可能不会很明显，但随着数据规模增长，性能的提升是非常显著的。

6.4.6 加载数据

至此，你已经了解了 Hive 风格的表格式和分区数据。如何把数据放到表里呢？这取决于谁掌握这些数据。让我们先假定你在 Presto 中创建表并用 Presto 加载数据。

```
CREATE TABLE hive.web.page_views (
  view_time timestamp,
  user_id bigint,
  page_url varchar,
  view_date date,
  country varchar
)
WITH (
  partitioned_by = ARRAY['view_date', 'country']
)
```

Presto 支持 `INSERT INTO ... VALUES`、`INSERT INTO ... SELECT` 和 `CREATE TABLE AS SELECT`

等语句，从而加载数据。虽然有 INSERT INTO 语句，但其作用是有限的，因为每个 INSERT 语句只创建一个文件或一行数据。这通常用于学习 Presto 的时候。

INSERT SELECT 和 CREATE TABLE AS 语句执行同样的功能。使用哪一个取决于是要加载到现有表中，还是在加载时创建表。以 INSERT SELECT 为例，你可能要从一个非分区的外部表中查询数据，然后在 Presto 中加载到一个分区表中：

```
presto:web> INSERT INTO page_views_ext SELECT * FROM page_views;
INSERT: 16 rows
```



上面的例子展示了在外部表中插入新数据。但默认情况下，Presto 不允许向外部表写入数据。要启用它，你需要在 catalog 配置文件中将 hive.non-managed-table-writes-enabled 设置为 true。

如果你熟悉 Hive，Presto 支持所谓的**动态分区**：当 Presto 检测到分区列值没有对应的目录时会创建它。

你也可以在 Presto 中创建一个外部分区表。假设在 S3 中一个带数据的目录结构如下所示：

```
...
s3://example-org/page_views/view_date=2019-01-14/...
s3://example-org/page_views/view_date=2019-01-15/...
s3://example-org/page_views/view_date=2019-01-16/...
...
```

通过以下语句创建外部表的定义：

```
CREATE TABLE hive.web.page_views (
  view_time timestamp,
  user_id bigint,
  page_url varchar,
  view_date date
)
WITH (
  partitioned_by = ARRAY['view_date']
)
```

之后从中查询：

```
presto:web> SELECT * FROM page_views;
 view_time | user_id | page_url | view_date
-----------+---------+----------+-----------
(0 rows)
```

发生了什么？即使我们知道其中有数据，HMS 也无法识别分区。如果你熟悉 Hive，就会知道用于自动发现所有分区的 MSCK REPAIR TABLE 命令。幸运的是，Presto 也有自己的命令来自动发现和添加分区：

```
CALL system.sync_partition_metadata(
  'web',
  'page_views',
  'FULL'
)
...
```

现在我们已经添加了分区，再试一下：

```
presto:web> SELECT * FROM page_views;
        view_time        | user_id | page_url | view_date
-------------------------+---------+----------+------------
 2019-01-25 02:39:09.987 |     123 | ...      | 2019-01-14
 ...
 2019-01-25 02:39:11.807 |     123 | ...      | 2019-01-15
 ...
```

另外，Presto 提供了手动创建分区的能力。这通常很麻烦，因为你必须使用命令逐个定义每个分区：

```
CALL system.create_empty_partition[w][x](
  'web',
  'page_views',
  ARRAY['view_date'],
  ARRAY['2019-01-14']
)
...
```

当你想通过 ETL 过程在 Presto 之外创建分区，然后想将新数据暴露给 Presto 时，添加空分区是非常有用的。

Presto 还支持删除分区，只需在 `DELETE` 语句的 `WHERE` 子句中指定分区列值即可。在这个例子中，数据本身保持不变，因为它是外部表。

```
DELETE FROM hive.web.page_views
WHERE view_date = DATE '2019-01-14'
```

需要强调的是，不一定要用 Presto 来管理表和数据。许多用户利用 Hive 或其他工具来创建和操作数据，而 Presto 只用于查询数据。

6.4.7 文件格式和压缩

Presto 支持 Hadoop/HDFS 中的许多常用文件格式，包括：

- ORC
- PARQUET
- AVRO
- JSON

- TEXTFILE
- RCTEXT
- RCBINARY
- CSV
- SEQUENCEFILE

Presto 使用的三种最常用的文件格式是 ORC、Parquet 和 Avro 数据文件。ORC、Parquet、RC Text 和 RC Binary 格式的读取组件在 Presto 中做了大量的性能优化。

HMS 中的元数据包含文件格式信息，以便 Presto 知道在读取数据文件时使用什么读取组件。在 Presto 中创建一个表时，数据类型被默认设置为 ORC，但可以在 `CREATE TABLE` 语句中的 `WITH` 属性里覆盖默认的数据类型：

```
CREATE TABLE hive.web.page_views (
  view_time timestamp,
  user_id bigint,
  page_url varchar,
  ds_date,
  country varchar
)
WITH (
  format = 'ORC'
)
```

catalog 中所有表的默认存储格式可以通过 catalog 属性文件中的 `hive.storage-format` 配置来设置。

默认情况下，Presto 使用 GZIP 压缩编解码器来编写文件。可通过在 catalog 属性文件中设置 `hive.compression-codec` 配置，将代码更改为使用 `SNAPPY` 或 `NONE`。

6.4.8 MinIO示例

MinIO 是一个兼容 S3 的轻量级分布式存储系统，你可以使用 Presto 和 Hive 连接器来访问它。如果你想更详细地了解它的使用，可以查看我们的示例项目。



如果你的 HDFS 开启了 Kerberos，你可以在 10.8 节中了解更多关于 Hive 连接器的配置。

6.5 非关系数据源

Presto 利用连接器来查询非关系数据源的变体。这些数据源通常叫作 NoSQL 系统，可以是键值存储、列存储、流处理系统、文档存储和其他系统。

其中一些数据源提供了类似 SQL 的查询语言，如 Apache Cassandra 的 CQL；另一些则只提供了特定的工具或 API 来访问数据，或者使用完全不同的查询语言，如 Elasticsearch 中使用的 DSL（domain specific language）查询。这些语言只具有有限的完备性并且没有标准化。

这些 NoSQL 数据源的 Presto 连接器允许你在这些系统上运行 SQL 查询，就像关系数据源一样。这使得你可以使用商业智能工具等应用程序，或者允许懂 SQL 的人查询这些数据源。这包括在这些数据源上使用 Join、聚合、子查询和其他高级 SQL 功能。

你将在第 7 章中了解到更多关于这些连接器的内容。

- NoSQL 系统，如 Elasticsearch 或 MongoDB（参见 7.5 节）。
- 流系统，如 Apache Kafka（参见 7.4 节）。
- 键值存储系统，如 Apache Accumulo（参见 7.2 节）和 Apache Cassandra（参见 7.3 节）。
- 用于 Apache HBase 的 Apache Phoenix 连接器（参见 7.1 节）。

我们先跳过它们，讨论一些比较简单的连接器。

6.6 Presto JMX连接器

JMX 连接器可以通过 catalog 属性文件 etc/catalog/jmx.properties 轻松配置使用。

```
connector.name=jmx
```

JMX 连接器公开了关于运行 Presto 协调器和工作节点的 JVM 的运行时信息。它使用 **Java 管理扩展**（Java management extensions，JMX），允许你使用 Presto 的 SQL 接口来访问可用信息。这对于监控和故障排除非常有用。

该连接器为历史数据、聚合数据提供了名为 `history` 的 schema，为最新数据提供了名为 `current` 的 schema，并且为元数据提供了名为 `information_schema` 的 schema。

最简单的学习方法就是用 Presto 语句来查看有哪些可用的表。

```
SHOW TABLES FROM jmx.current;
                Table
------------------------------------------------------------------
 com.sun.management:type=diagnosticcommand
 com.sun.management:type=hotspotdiagnostic
 io.airlift.discovery.client:name=announcer
 io.airlift.discovery.client:name=serviceinventory
 io.airlift.discovery.store:name=dynamic,type=distributedstore
 io.airlift.discovery.store:name=dynamic,type=httpremotestore
 ...
```

如你所见，表名使用 Java classpath，其包含指标所发出的类及其参数。这意味着，在 SQL 语句中引用表名时需要使用引号。一般来说，查看表中的可用列非常有用：

```
DESCRIBE jmx.current."java.lang:type=runtime";
 Column         |  Type    | Extra | Comment
------------------------+---------+-------+---------
 bootclasspath          | varchar |       |
 bootclasspathsupported | boolean |       |
 classpath              | varchar |       |
 inputarguments         | varchar |       |
 librarypath            | varchar |       |
 managementspecversion  | varchar |       |
 name                   | varchar |       |
 objectname             | varchar |       |
 specname               | varchar |       |
 specvendor             | varchar |       |
 specversion            | varchar |       |
 starttime              | bigint  |       |
 systemproperties       | varchar |       |
 uptime                 | bigint  |       |
 vmname                 | varchar |       |
 vmvendor               | varchar |       |
 vmversion              | varchar |       |
 node                   | varchar |       |
 object_name            | varchar |       |
(19 rows)
```

这使你可以很好地格式化信息：

```
SELECT vmname, uptime, node FROM  jmx.current."java.lang:type=runtime";
          vmname          | uptime  |     node
--------------------------+---------+-------------
 OpenJDK 64-Bit Server VM | 1579140 | ffffffff-ffff
(1 row)
```

请注意，此查询只返回一个节点，因为它运行在第 2 章所述的单协调器 / 工作节点的简单安装上。

JMX 连接器将大量有关 JVM 的信息暴露出来，其中包括 Presto 特定的信息。可以通过查看以 `presto` 开头的表来浏览可用的数据，例如 `DESCRIBE  jmx.current."presto.execution:name=queryexecution";`。

以下是其他一些值得查看的 `DESCRIBE` 语句：

```
DESCRIBE jmx.current."presto.execution:name=querymanager";
DESCRIBE jmx.current."presto.memory:name=clustermemorymanager";
DESCRIBE jmx.current."presto.failuredetector:name=heartbeatfailuredetector";
```

要了解更多使用 Web UI 监控 Presto 和其他相关方面的信息，你可以阅读第 12 章。

6.7 黑洞连接器

黑洞连接器可以很容易地在 catalog 属性文件（如 etc/catalog/blackhole.properties）中配置使用。

```
connector.name=blackhole
```

它作为任何数据的最终消费者，类似于 UNIX 操作系统中的 null 设备（/dev/null）。你可以把它作为从其他 catalog 中读取并插入数据的目标。因为它实际上不写入任何内容，所以你可以用它来衡量从这些 catalog 中读取数据的性能。

例如，你可以在 `blackhole` 中创建名为 `test` 的 schema，并从 `tpch.tiny` 数据集中创建一个表。然后从 tpch.sf3 中读取一个数据集，并将其插入 `blackhole` catalog 中。

```
CREATE SCHEMA blackhole.test;
CREATE TABLE blackhole.test.orders AS SELECT * from tpch.tiny.orders;
INSERT INTO blackhole.test.orders SELECT * FROM tpch.sf3.orders;
```

这个操作本质上是衡量从 `tpch` catalog 中读取性能，它读取了 150 万条订单记录，然后将其发送到 `blackhole`。使用其他模式（如 `tcph.sf100`）会增加数据集的大小。这可以用来评估你的 Presto 集群的性能。

可以用 RDBMS、对象存储或键值存储 catalog 进行类似的查询，帮助我们进行查询开发、性能测试及改进。

6.8 内存连接器

内存连接器可以通过配置 catalog 属性文件开启；例如，etc/catalog/memory.properties。

```
connector.name=memory
```

你可以像使用临时数据库一样使用内存连接器。所有的数据都存储在集群的内存中，停止集群就会销毁数据。当然，你也可以主动使用 SQL 语句来删除表中的数据，甚至删除整张表。

使用内存连接器对查询测试或临时存储非常有用。例如，在使用鸢尾花数据集时，我们将它当作一个外部数据源的简单替代物，参见 1.4.7 节。



虽然内存连接器对于测试和做小任务很有用，但对于大数据集和在生产环境中使用（尤其是数据分布在集群中时）**并不适合**。例如，数据可能会分布在不同的工作节点上，如果一个工作节点崩溃，就会导致该数据丢失。仅对临时数据可以使用内存连接器。

6.9 其他连接器

你现在知道，Presto 项目包含许多连接器，但有时你会遇到这样的情况：你需要为一个特定的数据源添加一个新的连接器。

好消息是，通常这不会有太大阻碍。Presto 团队和社区正在不断扩大可用连接器的列表，所以当你读到本书时，这个列表可能已经变得更长了。

连接器也可以从 Presto 项目以外的第三方获得。这包括其他社区成员和 Presto 用户编写的连接器，它们还没有贡献回来，或者由于种种原因不能贡献出来。

连接器也可以从数据库系统的商业供应商那里获得，因此，询问你想查询的系统的所有者或创建者是个好主意。Presto 社区中有商业供应商，如 Starburst，他们将 Presto 与支持服务和扩展程序打包在一起销售，其中也包括额外的连接器或改进过的连接器。

最后，你必须记住，Presto 是一个围绕着开源项目的开放社区。因此，你可以——我们也鼓励你——去查看现有连接器的代码，并根据需要创建新的连接器。理想状况下，你甚至可以参与 Presto 项目，贡献一个连接器给该项目，从而让版本维护和使用变得更简单。

6.10 小结

现在你已经了解了 Presto 可以访问各种数据源的强大功能。无论你访问的是什么数据源，Presto 都能让你用 SQL 和 SQL 驱动的工具查询数据。尤其是，你学习了关键的 Hive 连接器，它用于查询分布式存储，如 HDFS 或云存储系统。在第 7 章中，你可以详细了解其他一些使用广泛的连接器。

所有连接器的详细文档可以在 Presto 网站上找到，参见 1.4.1 节。如果没有找到想要的内容，你甚至可以与社区合作，创建你自己的连接器或增强现有的连接器。

第 7 章

高级连接器实例

第 6 章中，你学习了 Presto 提供的一些连接器以及如何配置它们，本章会把这些知识扩展到一些比较复杂的使用场景和连接器。这些连接器通常需要翻译来自底层数据源的存储模式和思想，而这些数据源并不容易映射到 SQL 和 Presto 的面向表的模型上。

若打算通过 Presto 连接并用 SQL 查询特定的系统，你可以直接跳到对应章节来了解更多。

- 7.1 节，用 Phoenix 连接到 Hbase。
- 7.2 节，键值存储连接器示例：Accumulo。
- 7.3 节，Apache Cassandra 连接器。
- 7.4 节，流系统连接器示例：Kafka。
- 7.5 节，文档存储连接器示例：Elasticsearch。

在了解这些连接器之后，你可以通过学习 7.6 节中的联邦查询和相关的 ETL 用法来进一步完善你的理解。

7.1　用Phoenix连接HBase

分布式、可扩展的大数据存储 Apache HBase 构建在 HDFS 之上，但用户并不一定通过 Hive 连接器访问低层的 HDFS。Apache Phoenix 项目提供了一个 SQL 层来访问 HBase，得益于 Presto Phoenix 连接器，用户可以像其他数据源一样从 Presto 访问 HBase 数据库。

和之前一样，你只需要一个 catalog 文件，比如 etc/catalog/bigtables.properties。

```
connector.name=phoenix
phoenix.connection-url=jdbc:phoenix:zookeeper1,zookeeper2:2181:/hbase
```

连接 URL 是指向数据库的 JDBC 连接串。它包括一个 Apache ZooKeeper 节点的列表，用于发现 HBase 节点。

Phoenix 的 schema 和表被映射到 Presto 的 schema 和表，可以用以下 Presto 语句来查看：

```
SHOW SCHEMAS FROM bigtable;
SHOW TABLES FROM bigtable.example;
SHOW COLUMNS FROM bigtable.examples.user;
```

现在你可以查询任何 HBase 表，并在下游工具中使用它们，就像从任何 Presto 的其他数据源取得数据一样。

使用 Presto 可以让你利用其水平扩展的性能优势来查询 HBase。你可以创建任何查询来访问 HBase 和其他 catalog，并且能够将 HBase 与其他数据源的数据结合起来进行联邦查询。

7.2　键值存储连接器示例：Accumulo

Presto 包含多个键值数据存储的连接器。**键值存储系统**用于管理记录存储的字典，并允许通过唯一的键来检索某条记录。想想散列表，其中的记录是通过一个键来检索的，这条记录可以是单值、多值甚至集合。

目前有几种键值存储系统。其中一个广泛使用的系统是开源、宽列存储数据库 Apache Cassandra，它有一个 Presto 连接器。你可以在 7.3 节中找到更多信息。

我们现在详细讨论另一个例子：Apache Accumulo。它是一个性能优异、应用广泛的开源键值存储，可以用 Presto 连接器进行查询，一般概念也适用于其他的键值存储。

受 Google 的 BigTable 的启发，Apache Accumulo 是一个有序的分布式键值存储，用于可扩展的存储和检索。Accumulo 在 HDFS 上存储按键排序的键值数据。

图 7-1 显示了 Accumulo 中，由行 ID、列和时间戳组成的一个三元组如何构成键。键首先按行 ID 和列以字典序升序排序，然后按时间戳降序排序。

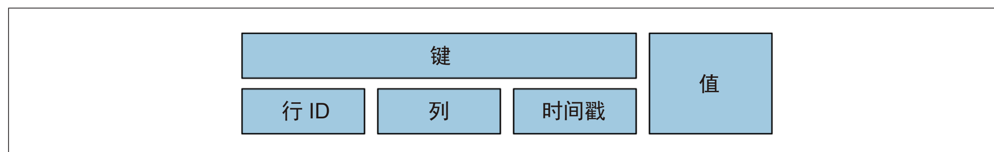


图 7-1：Accumulo 中的一个键值对

Accumulo 可以借助列族和位置组来进一步优化。大部分的优化对 Presto 来说是透明的，但了解 SQL 查询的访问模式可以帮助你优化 Accumulo 的表结构。这与为任何使用 Accumulo 的其他应用程序优化表一样。

表 7-1 展示了一个关系表的逻辑表示。

表7-1：Accumulo中数据的关系或逻辑视图

rowid	flightdate	flightnum	origin	dest
1234	2019-11-02	2237	BOS	DTW
5678	2019-11-02	133	BOS	SFO
⋮	⋮	⋮	⋮	⋮

因为 Accumulo 是键值存储，所以它在磁盘上存储这种数据表示方式与逻辑视图不同，如表 7-2 所示。这种非关系的存储方式使得不那么容易确定 Presto 如何从中读取数据。

表 7-2：Accumulo如何存储数据的视图

rowid	column	value
1234	flightdate:flightdate	2019-11-02
1234	flightnum:flightnum	2237
1234	origin:origin	BOS
1234	dest:dest	DTW
5678	flightdate:flightdate	2019-11-02
5678	flightnum:flightnum	133
5678	origin:origin	BOS
5678	dest:dest	SFO
⋮	⋮	⋮

Presto Accumulo 连接器负责将 Accumulo 数据模型映射到 Presto 可以理解的关系模型。

Accumulo 使用 HDFS 存储数据，并利用 ZooKeeper 管理表的元数据，如图 7-2 所示。

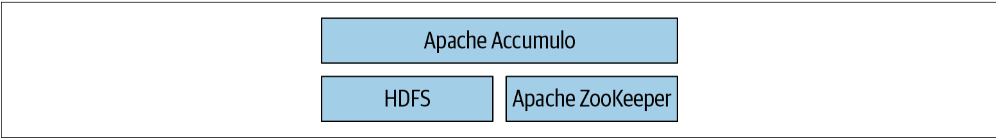


图 7-2：由分布式 Apache Accumulo、HDFS 和 Apache ZooKeeper 组成的 Accumulo 基本体系结构

就其核心而言，Accumulo 是一个分布式系统，它由一个主节点和多个 tablet 服务器组成，如图 7-3 所示。tablet 服务器包含并暴露 tablet，tablet 是一个表的水平分区片段。客户端直接连接到 tablet 服务器上，从而扫描需要的数据。

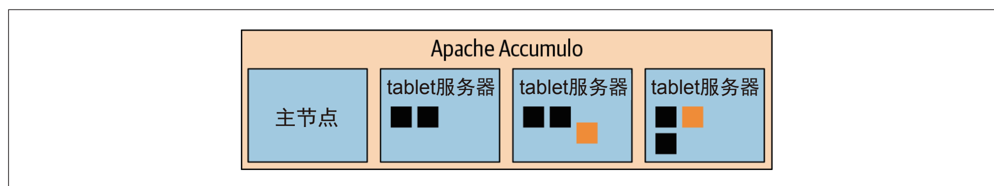


图 7-3：具有主节点和多个 tablet 服务器的 Accumulo 架构

就像 Accumulo 本身，Presto Accumulo 连接器也会用到 ZooKeeper。它从 Accumulo 使用的 ZooKeeper 实例中读取所有信息，如表、视图、表属性和列定义等。

下面来看看如何使用 Presto 从 Accumulo 中扫描数据。在 Accumulo 中，可以通过使用 `Scanner` 对象从表中读取键对。`Scanner` 从表中的某个特定的键开始读取，然后在另一个键或在表的末尾结束。`Scanner` 可以被配置为只读取所需列。回顾一下 RDBMS 连接器，只有所需列才会被添加到生成的 SQL 查询中，进而下推到数据库。

Accumulo 也有一个 `BatchScanner` 对象的概念，用于从 Accumulo 读取多个范围的数据。Accumulo 比 `Scanner` 高效，因为它能够使用多个工作节点与 Accumulo 通信，如图 7-4 所示。

用户首先将查询提交给协调器，协调器与 Accumulo 通信，以在元数据中确定分片。它通过从 Accumulo 可用的索引中寻找范围来确定分片。Accumulo 返回索引中的行 ID，Presto 将这些范围保存在分片中。如果不能使用索引，则对单个 tablet 中的所有范围使用分片。最后，工作节点使用这些信息连接到特定的 tablet 服务器，并从 Accumulo 中并行拉取数据，拉取数据是通过 `BatchScanner` 进行的。

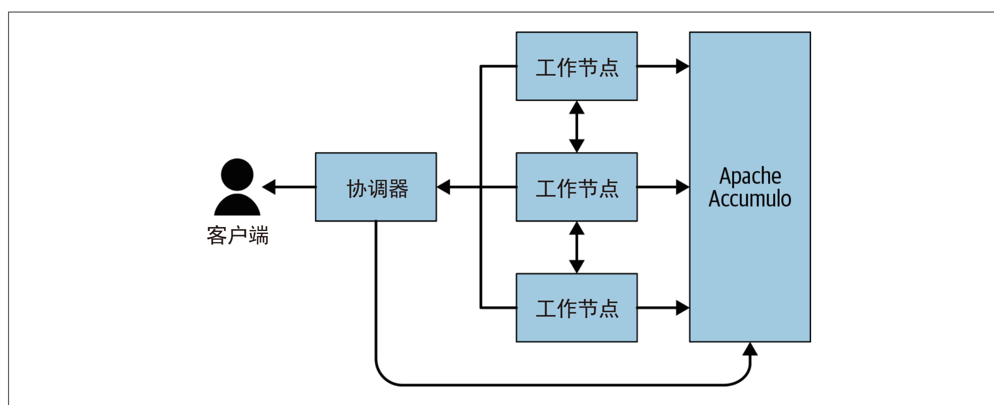


图 7-4：多个工作节点并行访问 Accumulo

一旦从工作节点那里拉取到数据，数据就会被放到 Presto 能够理解的关系格式中，剩下的处理工作便由 Presto 完成。在这种情况下，Accumulo 被当作数据存储，而 Presto 则提供了访问 Accumulo 数据的高级 SQL 接口。

如果你自己编写一个应用程序来从 Accumulo 中获取数据，那么你会写出类似下面的 Java 片段。先设置要扫描的范围，然后定义要拉取的列：

```
ArrayList<Range> ranges = new ArrayList<Range>();
ranges.add(new Range("1234"));
ranges.add(new Range("5678"));

BatchScanner scanner = client.createBatchScanner("flights", auths, 10);
scanner.setRangers(ranges);
scanner.fetchColumn("flightdate");
scanner.fetchColumn("flightnum");
scanner.fetchColumn("origin");

  for (Entry<Key,Value> entry : scanner) {
    // 填充为Presto格式
  }
}
```

裁剪不需要读取的列的概念与 RDBMS 连接器类似。Accumulo 连接器并不下推 SQL，而是使用 Accumulo API 来设置要拉取哪些列。

7.2.1 使用Presto Accumulo连接器

要使用 Accumulo，需要创建一个 catalog 属性文件（例如 etc/catalog/accumulo.properties），该文件引用 Accumulo 连接器，并配置了 Accumulo 访问方式，包括到 ZooKeeper 的连接：

```
connector.name=accumulo
accumulo.instance=accumulo
accumulo.zookeepers=zookeeper.example.com:2181
accumulo.username=user
accumulo.password=password
```

还是用之前那个航班的例子，让我们用 Presto 在 Accumulo 中创建一个表，可以使用 Presto CLI 或者通过 JDBC 连接到 Presto 的 RDBMS 管理工具：

```
CREATE TABLE accumulo.ontime.flights (
    rowid VARCHAR,
    flightdate VARCHAR,
    flightnum, INTEGER,
    origin VARCHAR
    dest VARCHAR
);
```

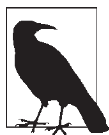
当你在 Presto 中创建这个表时，连接器实际上在 Accumulo 中创建了一个表，并在 ZooKeeper 中创建了关于该表的元数据。

也可以创建一个列族。Accumulo 中的**列族**是一种优化机制，用于应用程序经常一起访问的那些列。通过定义列族，Accumulo 可以安排列在磁盘上的存储方式，使经常共同访问

的列作为列族的一部分，存放在一起。如果想创建一张使用列族的表，则可以在 WITH 语句中将其指定为表属性。

```
CREATE TABLE accumulo.ontime.flights (
    rowid VARCHAR,
    flightdate VARCHAR,
    flightnum, INTEGER,
    origin VARCHAR
    dest VARCHAR
)
WITH
   column_mapping = 'origin:location:origin,dest:location:dest'
);
```

通过使用 column_mapping，你可以用列修饰符 origin 和 dest 来定义列族的位置，而这两个列的名称与 Presto 中的列名相同。



当不使用 column_mapping 表属性时，Presto 会自动生成与 Presto 列名相同的列族和列修饰符。可以通过在表上运行 DESCRIBE 命令来观察 Accumulo 列族和列修饰符。

Presto Accumulo 连接器支持 INSERT 语句。

```
INSERT INTO accumulo.ontime.flights VALUES
    (2232, '2019-10-19', 118, 'JFK', 'SFO');
```

这是一种方便的数据插入方式，但目前从 Presto 写入 Accumulo 的数据吞吐量很低。为了获得更好的性能，你需要使用原生的 Accumulo API。在 Presto 之外，Accumulo 连接器有工具可以完成性能更高的数据插入。可以在 Presto 文档中找到更多关于使用单独工具加载数据的信息。

我们在前面的例子中创建的表是一个**内部表**。Presto Accumulo 连接器支持内部表和外部表。这二者间的唯一区别是，删除外部表只删除元数据而不是数据本身。外部表可以让你创建已存在于 Accumulo 中的 Presto 表。此外，如果需要改变表的 schema，如添加一个列，你可以简单地删除表，然后在 Presto 中重新创建，而不会丢失数据。值得注意的是，当每一行的列集合不一定要一致时，Accumulo 可以支持这种表结构变更。

使用外部表需要多做一些工作，因为数据已经以特定的方式存储了。例如，在使用外部表时，必须使用 column_mapping 表属性。你必须在创建表时将 external 属性设置为 true。

```
CREATE TABLE accumulo.ontime.flights (
    rowid VARCHAR,
    flightdate VARCHAR,
    flightnum, INTEGER,
    origin VARCHAR
    dest VARCHAR
```

```
)
WITH
    external = true,
    column_mapping = 'origin:location:origin,dest:location:dest'
);
```

7.2.2 Accumulo中的谓词下推

在 Accumulo 连接器中，Presto 可以利用 Accumulo 内置的二级索引。为了实现这一点，Accumulo 连接器需要在每个 tablet 服务器上建立一个自定义的服务器端迭代器。该迭代器以 JAR 文件的形式分发，你必须将其复制到每个 tablet 服务器上的 $ACCUMULO_HOME/lib/ext 目录中。你可以在 Presto 文档中找到具体做法。

Accumulo 中的索引用于查询行 ID，行 ID 用来读取实际表中的值。让我们来看一个例子。

```
SELECT flightnum, origin
FROM flights
WHERE flightdate BETWEEN DATE '2019-10-01' AND 2019-11-05'
AND origin = 'BOS';
```

在没有索引的情况下，Presto 会从 Accumulo 中读取整个数据集，然后在 Presto 内部进行过滤。要读取的 Accumulo 范围被分配给各个工作节点，这里的范围是整个 tablet 的范围。如果有索引（如表 7-3 中的示例索引），处理范围的数量就可以大大减少。

表7-3：航班表上的索引示例

2019-08-10	flightdate_flightdate:2232	[]
2019-10-19	flightdate_flightdate:2232	[]
2019-11-02	flightdate_flightdate:1234	[]
2019-11-02	flightdate_flightdate:5478	[]
2019-12-01	flightdate_flightdate:1111	[]
SFO	origin_origin:2232	[]
BOS	origin_origin:3498	[]
BOS	origin_origin:1234	[]
BOS	origin_origin:5678	[]
ATL	origin_origin:1111	[]
⋮	⋮	⋮

协调器使用 `WHERE` 子句的过滤条件 `flightdate BETWEEN DATE '2019-10-01' AND 2019-11-05' AND origin = 'BOS'` 来扫描索引以获得表中的行 ID，然后将这些行 ID 打包成分片，工作节点在随后访问 Accumulo 中的数据时会使用。在我们的例子中，`flightdate` 和 `origin` 上有二级索引，我们收集了 {2232, 1234, 5478} 和 {3498, 1234, 5678} 两个集合的行 ID。把每个索引中提取出的行 ID 做交集，可以得出我们只需要扫描行 ID {1234，5678}。然后将这

个范围放入分片中，由工作节点进行处理，这样就可以直接访问各个值，数据详细视图如表 7-4 中所示。

表7-4：出发地、目的地及其他值的详细视图

Rowid	Column	Value
2232	flightdate:flightdate	2019-10-19
2232	flightnum:flightnum	118
2232	origin:origin	JFK
2232	dest:dest	SFO
1234	flightdate:flightdate	2019-11-02
1234	flightnum:flightnum	2237
1234	origin:origin	BOS
1234	dest:dest	DTW
5678	flightdate:flightdate	2019-11-02
5678	flightnum:flightnum	133
5678	origin:origin	BOS
5678	dest:dest	SFO
3498	flightdate:flightdate	2019-11-10
3498	flightnum:flightnum	133
3498	origin:origin	BOS
3498	dest:dest	SFO
⋮	⋮	⋮

为了利用谓词下推的优势，需要在想要下推谓词的列上建立索引。通过 Presto 连接器，可以使用 `index_columns` 表属性在列上建立索引。

```
CREATE TABLE accumulo.ontime.flights (
    rowid VARCHAR,
    flightdate VARCHAR,
    flightnum, INTEGER,
    origin VARCHAR
    dest VARCHAR
)
WITH
    index_columns = 'flightdate,origin'
);
```

在本节中，你学习了键值存储以及在 Presto 中如何使用标准 SQL 来查询它。接下来看另一个可以从 Presto 中受益的、使用更广泛的系统：Apache Cassandra。

7.3 Apache Cassandra连接器

Apache Cassandra 是一个分布式的、支持海量数据的宽列存储。Cassandra 的容错架构和线性扩展性使其得到了广泛的应用。

在 Cassandra 中处理数据的典型方法是使用为 Cassandra 设计的自定义查询语言：**Cassandra 查询语句**（Cassandra Query Language，CQL）。虽然 CQL 从表面上看很像 SQL，但它实际上缺少 SQL 的许多有用功能，如 Join。总的来说，它和 SQL 相当不同，无法直接使用依赖 SQL 的标准工具。

然而，借助 Cassandra 连接器，你可以对 Cassandra 中的数据进行 SQL 查询。最小配置就是一个简单的 catalog 文件，比如 etc/catalog/sitedata.properties。这里的 Cassandra 集群记录了一个网站上所有的用户交互，例如：

```
connector.name=cassandra
cassandra.contact-points=sitedata.example.com
```

有了这个简单的配置，用户就可以查询 Cassandra 中的数据。Cassandra 中的任何键空间（例如 `cart`）都会在 Presto 中作为一个 schema 暴露出来，现在可以用普通的 SQL 查询表了，比如 `users`：

```
SELECT * FROM sitedata.cart.users;
```

该连接器支持许多配置属性，可以让你配置调整 Cassandra 集群的 catalog，以及启用认证和连接的 TLS 等。

7.4　流式系统连接器示例：Kafka

流式系统和发布 – 订阅（pub/sub）系统旨在处理实时数据源。例如，LinkedIn 创造的 Apache Kafka 便是一个高吞吐量、低时延的平台。发布者将消息写到 Kafka 上，供订阅者消费。这样的系统一般用于系统之间的数据管道。Presto Kafka 连接器用于从 Kafka 中读取数据。目前你不能使用该连接器来发布数据。

借助该连接器，可以使用 SQL 查询 Kafka 主题上的数据，甚至可以将其与其他数据进行 Join 操作。Presto 的典型使用场景是，对实时 Kafka 主题流的 ad-hoc 查询，以检查和更好地了解当前系统的状态和数据流。使用 Presto 可以让数据分析师和其他用户更容易地访问 Kafka，因为他们通常不具备任何 Kafka 的背景知识，但知道如何编写 SQL 查询。

Presto 与 Kafka 的另一个不太常见的使用场景是从 Kafka 迁移数据。使用 `CREATE TABLE AS` 或 `INSERT SELECT` 语句，可以从 Kafka 主题中读取数据，使用 SQL 对数据进行变换，然后将其写入 HDFS、S3 或其他存储。

因为 Kafka 是一个流式系统，所以随着新数据的到来，暴露的主题会不断改变其内容。使用 Presto 查询 Kafka 主题时必须考虑到这一点。使用 Presto 将数据迁移到 HDFS 或其他具有永久存储的数据库系统中，可以保存流过 Kafka 主题的数据。

一旦数据在目标数据库或存储中永久可用，Presto 就可以用来将其暴露给分析工具，如 Apache Superset，参见 11.1 节。

使用 Kafka 连接器和其他连接器一样。创建一个使用 Kafka 连接器的 catalog 文件（例如 etc/catalog/trafficstream.properties），配置其他必要的细节，并指向你的 Kafka 集群：

```
connector.name=kafka
kafka.table-names=web.pages,web.users
kafka.nodes=trafficstream.example.com:9092
```

现在，Kafka 里的 `web.pages` 和 `web.user` 主题都可以在 Presto 中以表的形式出现。在任何时候，该表都会显示出整个 Kafka 主题和当前主题中的所有消息。主题中的每一条消息都会在 Presto 的表中显示为一行。这些数据现在可以在 Presto 上使用 catalog、schema 和表名进行 SQL 查询。

```
SELECT * FROM trafficstream.web.pages;
SELECT * FROM trafficstream.web.users;
```

本质上，你可以通过简单的 SQL 查询实时查看 Kafka 主题上的数据。

如果想要将数据迁移到另一个系统，如 HDFS catalog，则可以从一个简单的 `CREATE TABLE AS`（CTAS）命令开始：

```
CREATE TABLE hdfs.web.pages
WITH (
   format = 'ORC',
   partitioned_by = ARRAY['view_date']
)
AS
SELECT *
FROM trafficstream.web.pages;
```

一旦有了这个表，你就可以定期运行 `INSERT` 查询，向其中插入更多的数据：

```
INSERT INTO hdfs.web.pages
SELECT *
FROM trafficstream.web.pages;
```

为了避免重复复制，你可以跟踪 Kafka 中的一些由连接器暴露的内部列，具体来说，可以使用 `_partition_id`、`_partition_offset`、`_segment_start`、`_segment_end` 和 `_segment_count` 列。定期运行查询的具体设置取决于 Kafka 中关于删除消息的配置以及运行查询的工具，例如 Apache Airflow，参见 11.3 节。

Kafka 主题以表的形式暴露，其映射和包含的消息可以用一个 JSON 文件来定义，每个主题的映射可以在 etc/kafka/schema.tablename.json 中定义。具体到前面的例子，可以在 etc/kafka/web.pages.json 中定义映射。

不同主题内的 Kafka 消息可能使用不同的格式，Kafka 连接器包括了最常见格式的解码器，如 Raw、JSON、CSV 和 Avro 等。

有关配置属性、映射和其他内部列的详细信息可在 Presto 文档中找到，参见 1.4.3 节。

将 Presto 与 Kafka 一起使用，既可以为 Kafka 的数据流开启新的分析与洞察能力，也定义了 Presto 另一个有价值的用途。另一个 Presto 支持的类似用途的流处理系统是 Amazon Kinesis[1]。

7.5 文档存储连接器示例：Elasticsearch

Presto 包含一些知名文档存储系统的连接器，如 Elasticsearch 或 MongoDB。这些系统支持存储和检索 JSON 文档形式的信息。Elasticsearch 更适合于索引和搜索文档，而 MongoDB 则是一个通用的文档存储。

7.5.1 概述

Presto 连接器允许用户使用 SQL 访问这些系统并查询其中的数据，即使这些系统没有原生 SQL 访问。

Elasticsearch 集群经常用于存储日志数据或其他事件流，以便长期存储，甚至永久存储。这些数据集都是系统在运行中和各种场景下产生的日志数据，往往体量非常大，这些日志可以帮我们更好地了解系统。

Elasticsearch 和 Presto 是一个强大且性能优异的组合，因为这两个系统都可以横向扩展。Presto 通过将一个查询分解并在集群中的许多工作节点上运行来进行扩展。

Elasticsearch 通常在自己的集群上运行，也可以横向扩展。它可以跨多个节点对索引分片，并以分布式方式运行任何搜索操作。Elasticsearch 集群的性能调优是一个独立的话题，需要了解搜索索引中的文档数量、集群中的节点数量、副本集合以及分片配置等细节。

然而，从客户端的角度来看（也即从 Presto 的角度来看），这一切都是透明的，只需用服务器的 URL 来暴露 Elasticsearch 集群即可。

7.5.2 配置和使用方法

配置 Presto 访问 Elasticsearch 是通过创建一个 catalog 文件（如 etc/catalog/search.properties）来进行的。

```
connector.name=elasticsearch
elasticsearch.host=searchcluster.example.com
```

注 1：最新支持的流处理系统还有 Apache Pulsar。——译者注

这个配置依赖于端口、schema 以及其他细节的默认值，但对于查询集群来说已经足够了。该连接器开箱即支持 Elasticsearch 的众多数据类型。它可以自动分析每个索引，将每个索引配置为表，在名为 `default` 的 schema 中暴露该表，创建必要的嵌套结构和行类型，并在 Presto 中全部暴露出来。索引中的任何文档都会自动对应到 Presto 的表结构中。例如，在 catalog 的 `default` schema 中，一个名为 `server` 的索引会自动成为一张表，你可以在 Presto 中查询该结构的更多信息：

```
DESCRIBE search.default.server;
```

用户可以立即开始查询索引。可以用 Information Schema 或 `DESCRIBE` 命令来了解每个索引/schema 所创建的表和字段。

Elasticsearch 的 schema 中的字段通常包含多个值，类似数组。如果自动检测不能满足要求，可以添加一个字段属性来定义索引映射。此外，`_source` 隐藏字段包含了来自 Elasticsearch 的源文档，如果需要的话，可以使用 JSON 文档解析功能（参见 9.15 节）以及集合数据类型（参见 8.9.1 节）。在处理 Elasticsearch 集群中的文档时，这些功能通常会有帮助，因为这些文档主要是 JSON 文档。

在 Elasticsearch 中，你可以将一个或多个索引中的数据（包括过滤后的数据）暴露成别名（`alias`）。Presto 连接器支持使用别名，并像其他索引一样以表的形式暴露出来。

7.5.3 查询处理

当你从 Presto 发出查询到 Elasticsearch 时，除了 Elasticsearch 本身的集群，Presto 也会利用其自身的集群架构来进一步提升性能。

Presto 可以查询 Elasticsearch 以了解所有的 Elasticsearch 分片（shard），之后在创建查询计划时使用这些信息。它将查询分为多个单独的切片（split），这些切片会并发地向特定的分片发起单独的请求。一旦结果返回，就会在 Presto 中进行合并，然后返回给用户。这意味着，Presto 与 Elasticsearch 结合后不仅可以使用 SQL 进行查询，而且比 Elasticsearch 本身的性能更好。

另外需要注意的是，有一种典型的 Elasticsearch 集群，它运行在负载均衡器后台，只通过 DNS 主机名暴露出来，而上述提到的单独连接到特定数据分片的行为也同样适用于这样的集群。

7.5.4 全文搜索

Elasticsearch 连接器的另一个强大功能是支持全文搜索。它允许你在 Presto 发出的 SQL 查询中使用 Elasticsearch 查询字符串。

例如，想想有一个索引包含某个网站所有博客文章，这些文档存储在 `blogs` 索引中。也许这些文章由许多字段组成，如 `title`、`intro`、`post`、`summary` 和 `authors` 等。借助全文搜索，你就可以写一个简单的查询，在所有字段中搜索某个特定的词，如 `presto`：

```
SELECT * FROM "blogs: +presto";
```

Elasticsearch 的查询字符串语法支持对不同的搜索词进行加权，以及其他适用于全文搜索的功能。

7.5.5 总结

Amazon Elasticsearch Service 特有的一个强大功能是支持 AWS 身份和访问管理。有关此配置的更多详细信息，以及使用 TLS 和 Presto 文档中其他技巧确保与 Elasticsearch 集群的连接，参见 1.4.2 节。

使用 Presto 与 Elasticsearch，你可以通过一些强大的、可以支持 SQL 的工具来分析索引中的丰富数据。你可以手动编写查询，也可以利用丰富的分析工具，这让你更好地理解集群中的数据。

利用 Presto MongoDB 连接器将 MongoDB 连接到 Presto 也具有类似的优势。

7.6 Presto中的联邦查询

在阅读了 1.3 节中关于 Presto 的所有使用场景，并了解了 Presto 中所有的数据源和可用的连接器之后，你现在可以了解更多关于 Presto 中联邦查询的信息了。**联邦查询**是指访问多个数据源中数据的查询。

这种查询可用于将来自多个 RDBMS 数据库的内容和信息结合在一起，比如将运行在 PostgreSQL 上的企业后台应用程序数据库和 MySQL 上的 Web 应用程序数据库相结合，也可以将 PostgreSQL 上的数据仓库与用同样运行在 PostgreSQL 或其他地方的源数据相结合来查询。然而，更强大的例子是，可以将来自 RDBMS 的查询与运行在其他非关系系统上的查询结合起来。数据仓库中的数据与对象存储中的数据可以结合起来，后者存储着海量的 Web 应用程序数据。还可以将数据与键值存储或 NoSQL 数据库中的内容相关联。对象存储数据湖突然可以用 SQL 暴露出来了，这些信息便成了你更好地理解整体数据的基础。

联邦查询可以帮助你真正地理解这些系统中的数据。

在下面的例子中，你将了解到将分布式存储中的数据与 RDBMS 中的数据联接起来的使用场景。必要的设置信息参见 1.4.8 节。

有了这些数据，你可以通过 SQL 查询提出诸如“飞机每年的平均延误时间是多少”这样的问题。

```
SELECT avg(depdelayminutes) AS delay, year
FROM flights_orc
GROUP BY year
ORDER BY year DESC;
```

还可以问："2 月某周中飞离波士顿的最佳日子是哪一天？"

```
SELECT dayofweek, avg(depdelayminutes) AS delay
FROM flights_orc
WHERE month=2 AND origincityname LIKE '%Boston%'
GROUP BY dayofmonth
ORDER BY dayofweek;
```

因为多数据源和联邦查询的概念是 Presto 的一部分，所以我们鼓励你建立一个环境并探索数据，这些查询能够启发你自己创建更多的查询。我们使用航空公司数据上的两个示例分析查询来演示 Presto 中的查询联合。我们提供的设置使用存储在 S3 中的数据，并配置 Presto 使用 Hive 连接器。但是，如果你愿意，也可以将数据存储在 HDFS、Azure Storage 或 Google Cloud Storage 中，并使用 Hive 连接器来查询数据。

在第一个示例查询中，我们希望 Presto 从 HDFS 中的数据中返回航班最多的前 10 家航空公司：

```
SELECT uniquecarrier, count(*) AS ct
FROM flights_orc
GROUP BY uniquecarrier
ORDER BY count(*) DESC
LIMIT 10;
 uniquecarrier |    ct
---------------+----------
 WN            | 24096231
 DL            | 21598986
 AA            | 18942178
 US            | 16735486
 UA            | 16377453
 NW            | 10585760
 CO            |  8888536
 OO            |  7270911
 MQ            |  6877396
 EV            |  5391487
(10 rows)
```

虽然以上查询为我们提供了航班数最多的前 10 家航空公司的结果，但这需要你理解 `uniquecarrier` 列中缩写的含义。如果有一个列可以提供完整的航空公司名称而不是缩写就更好了，但我们查询的航空公司数据源中并不包含这样的信息。如果有另一个数据源包含这些信息，我们就可以将该数据源与之结合起来，从而返回更易于理解的结果。

让我们来看看另一个例子。在这里，我们希望 Presto 能返回出发次数最多的前 10 个机场：

```
SELECT origin, count(*) AS ct
FROM flights_orc
GROUP BY origin
ORDER BY count(*) DESC
LIMIT 10;
  origin |  ct
--------+---------
 ATL    | 8867847
 ORD    | 8756942
 DFW    | 7601863
 LAX    | 5575119
 DEN    | 4936651
 PHX    | 4725124
 IAH    | 4118279
 DTW    | 3862377
 SFO    | 3825008
 LAS    | 3640747
(10 rows)
```

和前面的查询一样，结果需要一些领域的专业知识。例如，你需要了解 `origin` 这一列所包含的机场代码。这些代码对于缺乏专业知识的人来说是没有意义的。

让我们通过与关系数据库中的附加数据相结合的方式来增强结果。在我们的例子中使用的是 PostgreSQL，但类似的步骤也适用于任何关系数据库。

与航空公司数据一样，我们的 GitHub 仓库包含了在关系数据库中创建和加载表的设置，以及如何配置 Presto 连接器来访问它。我们配置 Presto，使之可以从 PostgreSQL 数据库中查询额外的航空公司数据。PostgreSQL 中的表 `carrier` 提供了航空公司代码与航空公司名称之间的映射。你可以在我们的第一个查询示例中使用这些附加数据。

我们来看看 PostgreSQL 中的表 `carrier`。

```
SELECT * FROM carrier LIMIT 10;
 code |                 description
------+----------------------------------------------
 02Q  | Titan Airways
 04Q  | Tradewind Aviation
 05Q  | Comlux Aviation, AG
 06Q  | Master Top Linhas Aereas Ltd.
 07Q  | Flair Airlines Ltd.
 09Q  | Swift Air, LLC
 0BQ  | DCA
 0CQ  | ACM AIR CHARTER GmbH
 0GQ  | Inter Island Airways, d/b/a Inter Island Air
 0HQ  | Polar Airlines de Mexico d/b/a Nova Air
(10 rows)
```

这个表包含了代码（`code`）列和描述（`description`）列。利用这些信息，我们可以将第一个例子查询 `flights_orc` 表改为与 PostgreSQL 的 `carrier` 表中的数据进行 Join 操作。

```
SELECT f.uniquecarrier, c.description, count(*) AS ct
FROM hive.ontime.flights_orc f,
    postgresql.airline.carrier c
WHERE c.code = f.uniquecarrier
GROUP BY f.uniquecarrier, c.description
ORDER BY count(*) DESC
LIMIT 10;
 uniquecarrier |       description        |    ct
---------------+--------------------------+----------
 WN            | Southwest Airlines Co.     | 24096231
 DL            | Delta Air Lines Inc.       | 21598986
 AA            | American Airlines Inc.     | 18942178
 US            | US Airways Inc.            | 16735486
 UA            | United Air Lines Inc.      | 16377453
 NW            | Northwest Airlines Inc.    | 10585760
 CO            | Continental Air Lines Inc. |  8888536
 OO            | SkyWest Airlines Inc.      |  7270911
 MQ            | Envoy Air                  |  6877396
 EV            | ExpressJet Airlines Inc.   |  5391487
(10 rows)
```

看！现在我们编写了一个单一的 SQL 查询，将 S3 和 PostgreSQL 中的数据进行了联邦查询，从而提供了更有价值的数据结果。我们无须知道或再去查找航空公司的代码，而是在结果中直接呈现了航空公司名称。

在查询中，引用表时必须使用完全限定名称[2]。当利用 `USE` 命令来设置默认的 catalog 和 schema 时，未限定的表名会链接到该 catalog 和 schema 中。但是，当你需要在外部查询 catalog 和 schema 时，表名必须是限定的；否则，Presto 会试图在默认的 catalog 和 schema 中找到它，并返回错误。总之，如果你是在默认 catalog 和 schema 中查询表，则无须对表名进行完全限定，但当引用默认范围之外的数据源时，建议使用完全限制名称。

接下来看一下 PostgreSQL 中的表 `airport`，在第二个例子中它是查询的一部分：

```
SELECT code, name, city FROM airport LIMIT 10;
 code |           name            |         city
------+---------------------------+----------------------
 01A  | Afognak Lake Airport      | Afognak Lake, AK
 03A  | Bear Creek Mining Strip   | Granite Mountain, AK
 04A  | Lik Mining Camp           | Lik, AK
 05A  | Little Squaw Airport      | Little Squaw, AK
 06A  | Kizhuyak Bay              | Kizhuyak, AK
 07A  | Klawock Seaplane Base     | Klawock, AK
 08A  | Elizabeth Island Airport  | Elizabeth Island, AK
 09A  | Augustin Island           | Homer, AK
 1B1  | Columbia County           | Hudson, NY
 1G4  | Grand Canyon West         | Peach Springs, AZ
(10 rows)
```

注 2：即带有 catalog、schema 的表名。——译者注

看一下 PostgreSQL 中的这个数据，你会发现 code 列可以用来与 flight_orc 表的第二个查询 Join。这样，你就可以利用 airport 表中的附加信息与查询结合来提供更多的细节：

```
SELECT f.origin, c.name, c.city, count(*) AS ct
FROM hive.ontime.flights_orc f,
    postgresql.airline.airport c
WHERE c.code = f.origin
GROUP BY origin, c.name, c.city
ORDER BY count(*) DESC
LIMIT 10;
 origin |                 name                     |        city          |   ct
--------+------------------------------------------+----------------------+---------
 ATL    | Hartsfield-Jackson Atlanta International | Atlanta, GA          | 8867847
 ORD    | Chicago OHare International              | Chicago, IL          | 8756942
 DFW    | Dallas/Fort Worth International          | Dallas/Fort Worth, TX | 7601863
 LAX    | Los Angeles International                | Los Angeles, CA      | 5575119
 DEN    | Denver International                     | Denver, CO           | 4936651
 PHX    | Phoenix Sky Harbor International         | Phoenix, AZ          | 4725124
 IAH    | George Bush Intercontinental/Houston     | Houston, TX          | 4118279
 DTW    | Detroit Metro Wayne County               | Detroit, MI          | 3862377
 SFO    | San Francisco International              | San Francisco, CA    | 3825008
 LAS    | McCarran International                   | Las Vegas, NV        | 3640747
(10 rows)
```

看！和第一个例子一样，我们可以通过跨两个不同的数据源的 Join，提供更多有意义的信息。这里可以添加机场的名称，而不是让用户依赖难以解释的机场代码。

通过这个联邦查询的小例子，你会发现，在 Presto 中可以将不同的数据源和集中的查询在一个地方进行组合，从而显著改进查询结果。我们的例子只改善了查询结果的外观并增强了其可读性，但在很多情况下，利用更丰富、更大的数据集，联邦查询将不同来源的数据组合在一起，可以让我们对数据有一个全新的理解。

我们已经从终端用户的角度浏览了一些联邦查询的例子，接下来讨论一下这个架构是如何工作的，这基于第 4 章介绍的关于 Presto 架构的一些概念。

Presto 能够协调查询中涉及的数据源之间的混合执行。在前面的例子中，我们在分布式存储和 PostgreSQL 上进行查询。对于通过 Hive 连接器访问的分布式存储，Presto 会直接读取数据文件，无论数据是在 HDFS、S3、Azure Blob Storage 还是在其他的分布式存储上。而对于关系数据库连接器，如 PostgreSQL 连接器，Presto 依靠 PostgreSQL 作为执行的一部分。让我们使用前面的查询来说明，但为了让它更有趣，我们添加一个新谓词，它引用了 PostgreSQL 的 airport 表中的一个列：

```
SELECT f.origin, c.name, c.city, count(*) AS ct
FROM hive.ontime.flights_orc f,
  postgresql.airline.airport c
WHERE c.code = f.origin AND c.state = 'AK'
GROUP BY origin, c.name, c.city
ORDER BY count(*) DESC
LIMIT 10;
```

这个逻辑查询计划类似于图 7-5。可以看到这个计划包括扫描 flights_orc 表和 airport 表。这两个输入都被送入 Join 算子中。但在将机场数据送入 Join 算子之前，我们会应用一个 Filter 算子，因为我们只想查看 Alaska 机场的结果。在 Join 之后，应用了聚合和分组操作。最后由 TopN 算子将 ORDER BY 和 LIMIT 一并完成。

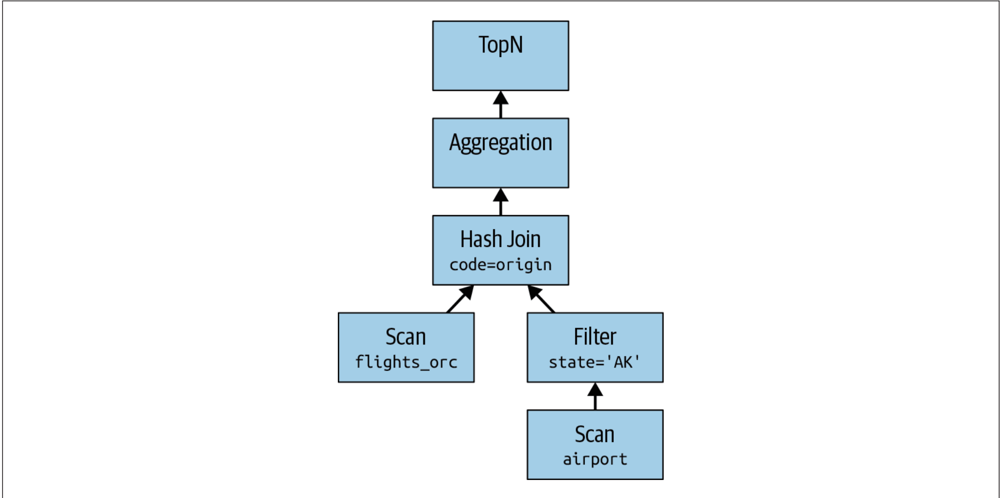


图 7-5：联邦查询的逻辑查询计划

为了从 PostgreSQL 中检索数据，Presto 通过 JDBC 发送一个查询。例如，最原始的方法是将下面的查询发送到 PostgreSQL：

```
SELECT * FROM airline.airport;
```

然而，Presto 比这更聪明，Presto 优化器会试图减少系统间传输的数据量。在这个例子中，Presto 只从 PostgreSQL 表中查询它所需要的列，同时将谓词下推到发送至 PostgreSQL 的 SQL 中。

因此，现在从 Presto 发送到 PostgreSQL 的查询会将更多的处理推送到 PostgreSQL：

```
SELECT code, city, name FROM airline.airport WHERE state = 'AK';
```

当到 PostgreSQL 的 JDBC 连接器返回数据到 Presto 时，Presto 会继续处理在 Presto 查询引擎中执行的部分。

一些比较简单的查询，如 SELECT * FROM public.airport，会完全下推到底层数据源中，如图 7-6 所示，这样查询的执行就发生在 Presto 之外，而 Presto 仅作为一个转发的角色。

目前，Presto 还不支持更复杂的 SQL 下推。例如，Presto 对只涉及 RDBMS 数据的聚合或 Join 才进行下推，以消除向 Presto 的数据传输。

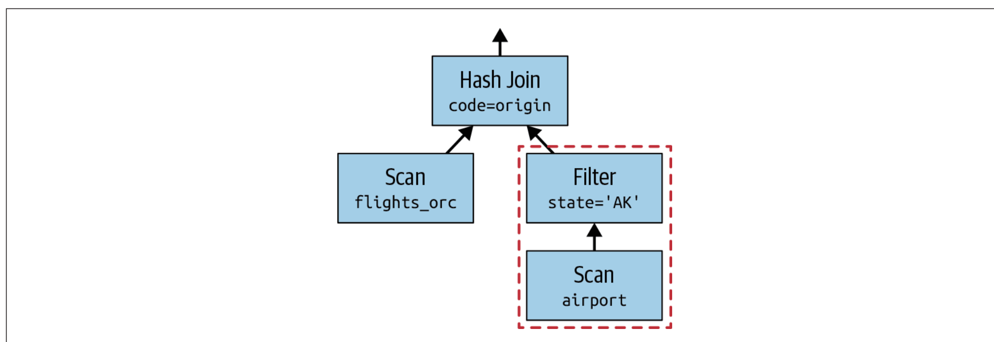


图 7-6：查询计划中的下推

7.7　ETL和联合查询

提取、转换、加载（extract, transform, load，简称 ETL）是一个术语，指从数据源复制数据并将其放到另一个数据源的技术。通常在准备目标数据的过程中会有一个从数据源转换数据的中间步骤。这个过程可能包括丢弃列、进行计算、过滤和清洗数据、Join 数据，以及进行预聚合等，这些都是为了让目标数据更适合查询。

Presto 并不是一个可以与商业解决方案相媲美的、成熟的 ETL 工具，但它可以通过避免 ETL 来提供一些帮助。因为 Presto 可以从数据源查询，所以可能不再需要移动数据。Presto 可以在数据所在的地方查询数据，以降低管理 ETL 过程的复杂性。

你可能仍需要做某些类型的 ETL 转换。也许你想查询预聚合的数据，或者你不想给底层系统带来更多的负载。通过使用 `CREATE TABLE AS` 或 `INSERT SELECT`，你可以将数据从一个数据源移动到另一个数据源。

在将 Presto 用于 ETL 场景的一大优势是支持关系数据库以外的其他数据源。

7.8　小结

你现在对 Presto 中的连接器有了很好的认识，是时候好好利用它们了。配置好你的 catalog，准备学习更多关于查询数据源的知识。

这就引出了我们的下一个主题——SQL 在 Presto 中的使用。SQL 知识对于你成功使用 Presto 至关重要，第 8 章和第 9 章会介绍所有你需要了解的内容。

第 8 章

在Presto中使用SQL

在安装和运行 Presto 之后，你先在 3.6 节中了解了 Presto 一流的 SQL 支持中所包含的核心功能。如有必要，可以回顾一下这部分内容。

在第 6 章中你知道了可以在 Presto 中用 SQL 查询很多数据源。

在本章中，你将深入了解 Presto 所支持 SQL 的细节，包括一组**数据定义语言**（data definition language，DDL）语句，它们用于创建和操作数据库对象，如 schema、表、列和视图。你还将学习更多的数据类型和 SQL 语句。在第 9 章中，你将学习到更多操作符和函数的高级用法。

总之，本章的目的不是作为 SQL 参考指南，而是展示 Presto 中 SQL 的能力。关于 Presto 中 SQL 的最新和最完整的信息，可以参考 Presto 官方文档，参见 1.4.2 节。



你可以通过使用 Presto CLI 或任何使用 JDBC 或 ODBC 驱动的应用程序来使用 SQL，这些都在第 3 章中讨论过。

连接器的影响

在 Presto 中执行的所有操作都依赖数据源的连接器及其对特定命令的支持。如果连接数据源的连接器不支持 `DELETE` 语句，执行 `DELETE` 就会失败。

此外，创建的连接通常使用特定的用户或其他授权机制，即存在特定的限制。如果用户没有除读取数据以外的权限，或者甚至只能从特定的 schema 中读取数据，那么其他操作（如删除数据或写入新数据）就会执行失败。

8.1 Presto语句

在使用 Presto 深入查询数据之前，知道哪些数据可用、在什么地方、是什么类型很重要。你可以通过 **Presto 语句**（Presto statement）收集这些信息。Presto 语句可以查询系统表和源数据，以获取 catalog 和 schema 等信息。这些语句与所有 SQL 语句在相同的上下文工作。

语句中的 `FROM` 和 `FOR` 子句需要输入一个完全限定的表、catalog 或 schema，除非之前使用 `USE` 设置了默认值。

`LIKE` 子句可以用来限制结果，其使用的 schema 匹配语法类似于 SQL 中的 `LIKE` 命令。

[] 中的命令部分是可选的。以下是 Presto 可用的一些语句。

`SHOW CATALOGS [ LIKE pattern ]`
: 列出可用的 catalog。

`SHOW SCHEMAS [ FROM catalog ] [ LIKE pattern ]`
: 列出 catalog 中的 schema。

`SHOW TABLES [ FROM schema ] [ LIKE pattern ]`
: 列出一个 schema 中的表。

`SHOW FUNCTIONS`
: 显示所有可用 SQL 函数的列表。

`SHOW COLUMNS FROM table` 或 `DESCRIBE table`
: 列出表中的列及其数据类型和其他属性。

`USE catalog.schema` 或 `USE schema`
: 更新会话以使用指定的 catalog 和 schema 作为默认值。如果没有指定 catalog，则使用当前 catalog 来解析 schema。

`SHOW STATS FOR table_name`
: 显示某个表中的数据规模和数量等统计信息。

`EXPLAIN`
: 生成查询计划，并详细列出各个步骤。

下面就来看一些常用的例子：

```
SHOW SCHEMAS IN tpch LIKE '%3%';
 Schema
---------
 sf300
 sf3000
 sf30000
(3 rows)

DESCRIBE tpch.tiny.nation;
  Column   |     Type     | Extra | Comment
-----------+--------------+-------+--------
 nationkey | bigint       |       |
 name      | varchar(25)  |       |
 regionkey | bigint       |       |
 comment   | varchar(152) |       |
(4 rows)
```

EXPLAIN 语句实际上比前面列表中给出的要更强大。下面是完整的语法：

```
EXPLAIN [ ( option [, ...] ) ] <query>
    options: FORMAT { TEXT | GRAPHVIZ | JSON}
             TYPE { LOGICAL | DISTRIBUTED | IO | VALIDATE }
```

可以使用 EXPLAIN 语句来显示查询计划：

```
EXPLAIN
SELECT name FROM tpch.tiny.region;
                               Query Plan
------------------------------------------------------------------------
 Output[name]
 │   Layout: [name:varchar(25)]
 │   Estimates: {rows: 5 (59B), cpu: 59, memory: 0B, network: 59B}
 └─ RemoteExchange[GATHER]
    │   Layout: [name:varchar(25)]
    │   Estimates: {rows: 5 (59B), cpu: 59, memory: 0B, network: 59B}
    └─ TableScan[tpch:region:sf0.01]
           Layout: [name:varchar(25)]
           Estimates: {rows: 5 (59B), cpu: 59, memory: 0B, network: 0B}
           name := tpch:name
```

这些计划信息能帮助你进行性能调优并更好地理解 Presto 如何处理查询。可以在第 4 章和第 12 章中了解更多关于此的信息。

EXPLAIN 的一个非常简单的使用场景是检查查询在语法上是否正确：

```
EXPLAIN (TYPE VALIDATE)
SELECT name FROM tpch.tiny.region;
```

8.2 Presto系统表

Presto 系统表不需要配置 catalog 文件，所有的 schema 和表都在 `system` catalog 中自动提供。

你可以使用 8.1 节中提到的语句来查询 schema 和表，以了解更多关于 Presto 运行实例的信息。可用的数据包括运行时、节点、catalog 等，这些信息可以让你更好地理解和使用 Presto。



Presto Web UI 是一个基于网页的用户界面，它显示了系统表的相关信息。更多详细信息参见 12.1 节。

系统表包含的 schema：

```
SHOW SCHEMAS IN system;
       Schema
--------------------
 information_schema
 jdbc
 metadata
 runtime
(4 rows)
```

为了进行查询调优，表 `system.runtime.queries` 和 `system.runtime.tasks` 是最有用的：

```
DESCRIBE system.runtime.queries;
            Column             |      Type      | Extra | Comment
------------------------------+----------------+-------+---------
 query_id                     | varchar        |       |
 state                        | varchar        |       |
 user                         | varchar        |       |
 source                       | varchar        |       |
 query                        | varchar        |       |
 resource_group_id            | array(varchar) |       |
 queued_time_ms               | bigint         |       |
 analysis_time_ms             | bigint         |       |
 distributed_planning_time_ms | bigint         |       |
 created                      | timestamp      |       |
 started                      | timestamp      |       |
 last_heartbeat               | timestamp      |       |
 end                          | timestamp      |       |
(13 rows)

DESCRIBE system.runtime.tasks;
         Column          |   Type    | Extra | Comment
------------------------+-----------+-------+---------
 node_id                | varchar   |       |
 task_id                | varchar   |       |
 stage_id               | varchar   |       |
 query_id               | varchar   |       |
 state                  | varchar   |       |
```

```
 splits                    | bigint    |        |
 queued_splits             | bigint    |        |
 running_splits            | bigint    |        |
 completed_splits          | bigint    |        |
 split_scheduled_time_ms   | bigint    |        |
 split_cpu_time_ms         | bigint    |        |
 split_blocked_time_ms     | bigint    |        |
 raw_input_bytes           | bigint    |        |
 raw_input_rows            | bigint    |        |
 processed_input_bytes     | bigint    |        |
 processed_input_rows      | bigint    |        |
 output_bytes              | bigint    |        |
 output_rows               | bigint    |        |
 physical_written_bytes    | bigint    |        |
 created                   | timestamp |        |
 start                     | timestamp |        |
 last_heartbeat            | timestamp |        |
 end                       | timestamp |        |
(23 rows)
```

上述表展示了一些底层数据，在 12.1 节中将详细介绍这些数据。system.runtime.queries 表提供了 Presto 中当前和历史查询的信息。system.runtime.tasks 表则提供了 Presto 中任务的底层细节。这与 Presto Web UI 中 Query Details 页面（查询详情页）的信息输出类似。

以下是几个查询系统表的有用例子。

列出 Presto 集群中的节点：

```
SELECT * FROM system.runtime.nodes;
```

显示所有失败的查询：

```
SELECT * FROM system.runtime.queries WHERE state='FAILED';
```

显示所有运行中的查询，包括它们的 query_id：

```
SELECT * FROM system.runtime.queries WHERE state='RUNNING';
```

系统表还提供了一种机制来终结运行中的查询。

```
CALL system.runtime.kill_query(query_id => 'queryId', message => 'Killed');
```

除了 Presto 运行时、集群以及工作节点等信息外，Presto 连接器还能够暴露其连接的数据源中的系统数据。例如，可以在 datalake catalog 中配置使用 6.4 节中讨论的 Hive 连接器。它可以自动地将 Hive 的系统表暴露出来：

```
SHOW TABLES FROM datalake.system;
```

结果中包含了诸如已用分区等方面的信息。

8.3 catalog

如第 6 章中所述，Presto catalog 代表在 catalog 属性文件中使用连接器配置的数据源。catalog 包含一个或多个 schema，而 schema 提供了表的集合。

例如，你可以配置一个 PostgreSQL catalog 来访问 PostgreSQL 上的关系数据库；也可以配置一个 JMX catalog，通过 JMX 连接器对 JMX 信息进行访问；还可以通过使用 Hive 连接器的 catalog 连接到 HDFS 对象存储数据源。当在 Presto 中运行一个 SQL 语句时，你是在一个或多个 catalog 上运行它。

多个 catalog 可以使用同一个连接器。例如，你可以创建两个独立的 catalog 来暴露两个运行在同一服务器上的 PostgreSQL 数据库。

在 Presto 中寻找一个表时，完全限定表名总是以 catalog 为根。例如，完全限定表名 `hive.test_data.test` 指的是 `hive` catalog、`test_data` schema 中的 `test` 表。

可以通过访问系统数据来查看 Presto 服务器中的可用 catalog 列表：

```
SHOW CATALOGS;
 Catalog
 ---------
 blackhole
 hive
 jmx
 postgresql
 kafka
 system
(6 rows)
```

catalog、schema 和表信息不由 Presto 存储，Presto 也没有自己的 catalog。连接器负责向 Presto 提供这些信息。通常情况下，这是通过从底层数据库查询 catalog 或通过连接器的其他配置来完成的。连接器处理这些请求，并在收到请求后返回相关信息。

8.4 schema

Presto 的 catalog 包含 schema。schema 包含表、视图和其他各种对象，是组织表的一种方式。catalog 和 schema 共同定义了一组可以查询的表。

当用 Presto 访问关系数据库（如 MySQL）时，schema 转化为目标数据库中的相同概念。而其他类型的连接器可能选择以底层数据源有意义的方式将表组织成 schema。连接器的实现决定了 schema 在 catalog 中的映射方式。例如，Hive 中的数据库在 Presto 的 Hive 连接器中是以 schema 的形式暴露出来的。

通常情况下，当你配置一个 catalog 时，schema 已经存在，但 Presto 也允许创建 schema 和

其他 schema 操作。

下面来看看创建 schema 的 SQL 语句：

```
CREATE SCHEMA [ IF NOT EXISTS ] schema_name
[ WITH ( property_name = expression [, ...] ) ]
```

`WITH` 子句可用于将属性关联到 schema。例如，对于 Hive 连接器，创建一个 schema 实际上就是在 Hive 中创建一个数据库。有时候，需要覆盖 `hive.metastore.warehouse.dir` 指定的数据库的默认位置：

```
CREATE SCHEMA hive.web
WITH (location = 's3://example-org/web/')
```

要获得可用的 schema 属性列表，你可以参考 Presto 的最新文档，或者在 Presto 中执行查询：

```
SELECT * FROM system.metadata.schema_properties;
-[ RECORD 1 ]-+------------------------------
catalog_name  | hive
property_name | location
default_value |
type          | varchar
description   | Base file system location URI
```

可以更改现有 schema 的名称：

```
ALTER SCHEMA name RENAME TO new_name
```

还支持删除一个 schema：

```
DROP SCHEMA [ IF EXISTS ] schema_name
```

当你不想让语句在 schema 不存在的情况下出错时，可以指定 `IF EXISTS`。在成功删除 schema 之前，你需要删除其中的表。有些数据库系统支持 `CASCADE` 关键字，表示用 `DROP` 语句将对象（如 schema）内的所有东西都删除。但是 Presto 现阶段并**不支持** `CASCADE`。

8.5 Information Schema

Information Schema 是 SQL 标准的一部分，并在 Presto 中作为一组视图，提供了关于 catalog 中的 schema、表、列、视图和其他对象的元数据。这组视图包含在一个名为 `information_schema` 的 schema 中。每个 Presto catalog 都有自己的 `information_schema`。像 `SHOW TABLES`、`SHOW SCHEMA` 等命令的结果与 Information Schema 中检索到的信息相同。

Information Schema 对于商业智能工具等第三方工具来说必不可少，其中的许多工具会查询 Information Schema，以便知道有哪些对象存在。

在每个连接器中，Information Schema 都有 8 个视图。对于某些不支持的功能（例如角色）的连接器，查询对应的 Information Schema 可能会产生一个不支持的错误：

```
SHOW TABLES IN system.information_schema;
      Table
 -----------------
 applicable_roles
 columns
 enabled_roles
 roles
 schemata
 table_privileges
 tables
 views
(8 rows)
```

注意，在你通过 Information Schema 查询 schema 中的表时，也会返回 Information Schema 自身拥有的表：

```
SELECT * FROM hive.information_schema.tables;
 table_catalog |    table_schema    |    table_name    | table_type
 --------------+--------------------+------------------+----------
 hive          | web                | nation           | BASE TABLE
 hive          | information_schema | enabled_roles    | BASE TABLE
 hive          | information_schema | roles            | BASE TABLE
 hive          | information_schema | columns          | BASE TABLE
 hive          | information_schema | tables           | BASE TABLE
 hive          | information_schema | views            | BASE TABLE
 hive          | information_schema | applicable_roles | BASE TABLE
 hive          | information_schema | table_privileges | BASE TABLE
 hive          | information_schema | schemata         | BASE TABLE
(9 rows)
```

此外，你可以使用 `WHERE` 子句来查询特定表的列：

```
SELECT table_catalog, table_schema, table_name, column_name
FROM hive.information_schema.columns
WHERE table_name = 'nation';
 table_catalog |    table_schema    |    table_name    |  column_name
--------------+--------------------+------------------+------------
 hive          | web                | nation           | regionkey
 hive          | web                | nation           | comment
 hive          | web                | nation           | nationkey
 hive          | web                | nation           | name
...
```

8.6 表

你已经了解了 catalog 和 schema，下面来学习一下 Presto 中表的定义。**表**是一组无序的行，它将数据组织成具有特定数据类型的命名列。和关系数据库中的表一样，Presto 的表由行、

列和这些列的数据类型所组成。从源数据到表的映射是由 catalog 定义的。

连接器的实现决定了表如何映射到 schema。例如，PostgreSQL 中的表是直接暴露在 Presto 中的，因为 PostgreSQL 原生支持 SQL 和表的概念。然而，要实现与其他系统的连接器，特别是当它们在设计上缺乏严格的表概念时，需要更多的创造力。例如，Apache Kafka 连接器将 Kafka topic 暴露为 Presto 中的表。

可以在 SQL 查询中使用完全限定名（*catalog-name.schema-name.table-name*）来访问表。

下面来看看在 Presto 中创建表的语句 `CREATE TABLE`：

```
CREATE TABLE [ IF NOT EXISTS ]
table_name (
  { column_name data_type [ COMMENT comment ]
  [ WITH ( property_name = expression [, ...] ) ]
  | LIKE existing_table_name [ { INCLUDING | EXCLUDING } PROPERTIES ] }
  [, ...]
)
[ COMMENT table_comment ]
[ WITH ( property_name = expression [, ...] ) ]
```

知道 SQL 的用户会十分熟悉上述语法。在 Presto 中，可选的 `WITH` 子句有一个重要的用途：其他的系统（如 Hive）扩展了 SQL 语言，使用户可以使用那些无法用标准 SQL 表达的逻辑或数据。遵循相同的方法违反了 Presto 尽可能地接近 SQL 标准的理念，也导致难以管理多种不同的连接器。因此 Presto 选择使用 `WITH` 语句指定表和列的属性，而不是扩展 SQL 语言。

表创建后便可使用标准 SQL 中的 `INSERT INTO` 语句。

例如，在鸢尾花数据集创建脚本中，首先创建一个表（参见 1.4.7 节），然后直接用查询插入值：

```
INSERT INTO iris (
  sepal_length_cm,
  sepal_width_cm,
  petal_length_cm,
  petal_width_cm,
  species )
VALUES
  ( ... )
```

如果数据是通过单独的查询获得的，则你可以在 `INSERT` 中使用 `SELECT`。假定你想把数据从 `memory` catalog 中复制到 PostgreSQL 中已有的一张表里：

```
INSERT INTO postgresql.flowers.iris
SELECT * FROM memory.default.iris;
```

`SELECT` 语句可以包括条件语句以及任何其他支持的语句。

8.6.1 表和列属性

让我们使用 6.4 节中介绍过的 Hive 连接器来创建一个表（见表 8-1），以学习 `WITH` 子句的用法。

表8-1：Hive连接器所支持的表属性

属性名称	属性描述
external_location	Hive 外部表所在文件系统的位置，例如，S3 或 Azure Blob Storage 上的位置
format	底层数据的文件存储格式，如 ORC、AVRO、PARQUET 等

我们使用表 8-1 中的属性通过 Presto 在 Hive 中创建一个表，这种创建方式与在 Hive 中创建表的方式相同。

我们先来看看 Hive 的语法：

```
CREATE EXTERNAL TABLE page_views(
  view_time INT,
  user_id BIGINT,
  page_url STRING,
  view_date DATE,
  country STRING)
 STORED AS ORC
 LOCATION 's3://example-org/web/page_views/';
```

与 Presto 中的 SQL 相比较：

```
CREATE TABLE hive.web.page_views(
  view_time timestamp,
  user_id BIGINT,
  page_url VARCHAR,
  view_date DATE,
  country VARCHAR
)
WITH (
  format = 'ORC',
  external_location = 's3://example-org/web/page_views'
);
```

可以看到，Hive DDL 扩展了 SQL 标准，而 Presto 使用属性达成同样的目的，因此符合 SQL 标准。

你可以查询 Presto 的系统元数据来列出可用的表属性：

```
SELECT * FROM system.metadata.table_properties;
```

要列出可用的列属性，可以运行下面的查询：

```
SELECT * FROM system.metadata.column_properties;
```

8.6.2 复制现有的表

可以将现有的表作为模板来新建一个表。LIKE 子句会创建一个与现有表具有相同列定义的表。默认情况下，表和列属性不会被复制。由于属性在 Presto 中很重要，我们建议在语法中使用 INCLUDING PROPERTIES 来复制它们。这个功能在使用 Presto 对数据进行某种类型的转换时非常有用：

```
CREATE TABLE hive.web.page_view_bucketed(
  comment VARCHAR,
  LIKE hive.web.page_views INCLUDING PROPERTIES
)
WITH (
  bucketed_by = ARRAY['user_id'],
  bucket_count = 50
)
```

使用 SHOW 语句来检查新建表的定义：

```
SHOW CREATE TABLE hive.web.page_view_bucketed;
                   Create Table
 ----------------------------------------------
 CREATE TABLE hive.web.page_view_bucketed (
    comment varchar,
    view_time timestamp,
    user_id bigint,
    page_url varchar,
    view_date date,
    country varchar
 )
 WITH (
    bucket_count = 50,
    bucketed_by = ARRAY['user_id'],
    format = 'ORC',
    partitioned_by = ARRAY['view_date','country'],
    sorted_by = ARRAY[]
 )
(1 row)
```

你可以将新表和原表进行比较：

```
SHOW CREATE TABLE hive.web2.page_views;
                   Create Table
--------------------------------------------------
 CREATE TABLE hive.web.page_views (
    view_time timestamp,
    user_id bigint,
    page_url varchar,
    view_date date,
    country varchar
 )
 WITH (
    format = 'ORC',
    partitioned_by = ARRAY['view_date','country']
 )
(1 row)
```

8.6.3 从查询结果中新建表

CREATE TABLE AS（CTAS）语句可用于新建包含 SELECT 查询结果的表。该表的列定义是根据查询结果的列数据动态创建的。该语句可用于创建临时表，或作为创建转换表过程的一部分：

```
CREATE TABLE [ IF NOT EXISTS ] table_name [ ( column_alias, ... ) ]
[ COMMENT table_comment ]
[ WITH ( property_name = expression [, ...] ) ]
AS query
[ WITH [ NO ] DATA ]
```

默认情况下，新表中填充了查询的结果。

CTAS 可用于转换表和数据。例如，你可以将未分区的 TEXTFILE 格式的数据加载到一个分区的 ORC 格式的表中：

```
CREATE TABLE hive.web.page_views_orc_part
WITH (
   format = 'ORC',
   partitioned_by = ARRAY['view_date','country']
)
AS
SELECT *
FROM hive.web.page_view_text
```

接下来的例子展示了如何从 page_views 表上的会话查询来创建新表：

```
CREATE TABLE hive.web.user_sessions
AS
SELECT user_id,
       view_time,
       sum(session_boundary)
         OVER (
           PARTITION BY user_id
           ORDER BY view_time) AS session_id
FROM (SELECT user_id,
             view_time,
             CASE
                WHEN to_unixtime(view_time) -
                     lag(to_unixtime(view_time), 1)
                        OVER(
                           PARTITION BY user_id
                           ORDER BY view_time) >= 30
             THEN 1
             ELSE 0
             END AS session_boundary
      FROM page_views) T
ORDER BY user_id,
         session_id
```



有时你只需要创建一个具有相同表定义而不包含数据的表。你可以通过在 CTAS 语句的末尾添加 `WITH NO DATA` 子句来实现。

8.6.4　修改表

`ALTER TABLE` 语句可以执行诸如重命名表、添加列、删除列或重命名列等操作：

```
ALTER TABLE name RENAME TO new_name

ALTER TABLE name ADD COLUMN column_name data_type
  [ COMMENT comment ] [ WITH ( property_name = expression [, ...] ) ]

ALTER TABLE name DROP COLUMN column_name

ALTER TABLE name RENAME COLUMN column_name TO new_column_name
```

需要注意的是，根据连接器和连接器的授权模型，默认可能不允许这些修改表的操作。例如，Hive 连接器就默认限制了这些操作。

8.6.5　删除表

使用 `DROP TABLE` 语句可以删除一个表：

```
DROP TABLE  [ IF EXISTS ] table_name
```

根据连接器的实现，这不一定会删除底层数据，你应该参考连接器文档以获得进一步的解释。

8.6.6　连接器对表操作的限制

目前，本章已经介绍了 Presto 支持的各种 SQL 语句。然而，这并不意味着 Presto 中的每一个数据源都支持所有的语句和语法，或者提供相同的语义。

连接器的实现和底层数据源的能力与语义很大程度上会影响 SQL 的支持程度。

如果你尝试使用连接器不支持的语句或操作，Presto 就会返回一个错误。例如，系统 schema 和表是用来暴露 Presto 系统信息的。在此 schema 下不支持创建表，因为这对于内部系统数据表没有意义。如果试图创建表，就会收到错误：

```
CREATE TABLE system.runtime.foo(a int);
Query failed: This connector does not support creating tables
```

8.7　视图

视图是基于 SQL 查询结果集的虚拟表。很多 RDBMS 对视图有良好的支持，但 Presto 不支持创建、编辑或删除视图。

Presto 把底层数据源中的视图当作表一样处理，这使得你可以将视图用于一些非常有用的目的。

- 将多个表格中的数据以更容易使用的视图形式暴露出来。
- 使用具有受限列或行的视图来限制可用数据。
- 方便地提供经过处理、转换的数据。

使用视图默认要求底层数据源对视图中的数据拥有完全的所有权，所以需要在底层数据源上创建并维护视图。因此，使用视图可以让你在几步之内将查询的处理下推到 RDBMS 中。

1. 发现 Presto 中运行的 SQL 查询有性能问题。
2. 通过查看执行 `EXPLAIN` 的计划来查找问题原因。
3. 意识到是特定的子查询造成的瓶颈。
4. 创建预处理子查询的视图。
5. 在你的 SQL 查询中使用该视图替换原表。
6. 享受性能提升。

8.8　会话信息和配置

当使用 Presto 时，所有的配置都在一个用户特定的上下文中维护，称为**会话**。该会话包含键值对，表示当前用户与 Presto 交互时所使用的诸多配置。

你可以使用 SQL 命令与这些信息进行交互。对于初学者来说，你可以直接查看当前的配置是什么，甚至可以使用 `LIKE` 模式来查看你感兴趣的选项：

```
SHOW SESSION LIKE 'query%';
```

此查询返回关于 `query_max_cpu_time`、`query_max_execution_time`、`query_max_run_time` 和 `query_priority` 的信息，包括当前值、默认值、数据类型（`INTEGER`、`BOOLEAN` 或 `VARCHAR`），以及属性的简要描述。

属性列表很长，里面包括了诸多影响 Presto 行为的配置选项，例如查询的内存和 CPU 限制、查询优化算法和是否使用基于代价的优化器等。

作为用户，你可以更改这些属性，这些属性只影响当前用户会话的表现。你可以为特定的查询或工作负载设置特定的选项，或者在将特定配置发布到全局的 config.properties 主配置文件之前先在会话中测试它们。

例如，你可以激活实验性的算法来使用 `collocated_join` 进行查询优化：

```
SET SESSION  collocated_join = true;
```

你可以通过以下语句确认是否设置成功：

```
SHOW SESSION LIKE 'collocated_join';
      Name      | Value | Default ...
----------------+-------+---------
 collocated_join | true  | false   ...
```

要撤销设置并恢复到默认值，你可以重置会话属性：

```
RESET SESSION  collocated_join;
```

8.9 数据类型

Presto 支持 SQL 标准描述的大部分数据类型，许多关系数据库也支持这些类型。本节将讨论 Presto 中支持的数据类型。

不是所有的 Presto 连接器都支持所有的 Presto 数据类型，并且 Presto 也可能不支持底层数据源中的所有类型，Presto 数据类型与底层数据源中的转换方式取决于连接器的实现。底层数据源可能不支持同名的类型，或者相同的类型可能有不同的命名方式。例如，MySQL 连接器将 Presto 的 `REAL` 类型映射到 MySQL 的 `FLOAT` 类型。

在某些情况下，数据类型需要进行转换。一些连接器将不支持的类型转换为 Presto 的 `VARCHAR` 类型（即源数据的字符串表示），或者在读取时完全忽略该列。具体细节可以在连接器的文档和源代码中找到。

让我们来看一下受到良好支持的数据类型列表。表 8-2 到表 8-6 描述了 Presto 中的数据类型，并提供了示例数据。

表8-2：布尔数据类型

类　　型	描　　述	示　　例
`BOOLEAN`	`true` 或 `false` 布尔值	`True`

表8-3：整数数据类型

类　　型	描　　述	示　　例
`TINYINT`	8 位有符号整数，最小值 -2^{7}，最大值 2^{7-1}	42
`SMALLINT`	16 位有符号整数，最小值 -2^{15}，最大值 2^{15-1}	42
`INTEGER, INT`	32 位有符号整数，最小值 -2^{31}，最大值 2^{31-1}	42
`BIGINT`	64 位有符号整数，最小值 -2^{63}，最大值 2^{63-1}	42

表8-4：浮点数据类型

类　型	描　述	示　例
REAL	32 位浮点数，遵循 IEEE 754 二进制浮点数运算标准	2.71828
DOUBLE	64 位浮点数，遵循 IEEE 754 二进制浮点数运算标准	2.71828

表8-5：固定精度数据类型

类　型	描　述	示　例
DECIMAL	固定精度小数	123456.7890

表8-6：字符串数据类型

类　型	描　述	示　例
VARCHAR 或 VARCHAR(*n*)	可变长度的字符串。当定义为 VARCHAR(*n*) 时，可以指定一个可选的正整数 *n* 代表最大字符长度值	"Hello World"
CHAR CHAR(*n*)	固定长度的字符串。当定义为 CHAR(*n*) 时，可以指定一个可选的正整数 *n* 代表字符长度。CHAR 等同于 CHAR(1)	"Hello World"

与 VARCHAR 不同，CHAR 总是分配 *n* 个字符。以下是你应该注意的一些特征和错误。

- 如果将一个少于 *n* 个字符的字符串转换为 CHAR(*n*)，则会在尾部添加空格。
- 如果将一个多于 *n* 个字符的字符串转换为 CHAR(*n*)，则会被截断，而不会报错。
- 如果你在表中插入一个比列中所定义长度长的 VARCHAR 或 CHAR，则会报错。
- 如果你在表中插入一个比列中所定义长度短的 CHAR，那么这个值会被填充空格，以匹配定义的长度。
- 如果你在表中插入一个比列中定义长度更短的 VARCHAR，则会保留字符串的确切长度。比较 CHAR 值时，前导和尾部的空格也被考虑在内。

下面的例子演示了这些行为：

```
SELECT length(cast('hello world' AS char(100)));
 _col0
 -----
   100
(1 row)

SELECT cast('hello world' AS char(15)) || '~';
      _col0
 ----------------
 hello world    ~
(1 row)

SELECT cast('hello world' AS char(5));
 _col0
 -------
 hello
(1 row)
```

```
SELECT length(cast('hello world' AS varchar(15)));
 _col0
 -------
    11
(1 row)

SELECT cast('hello world' AS varchar(15)) || '~';
    _col0
 --------------
 hello world~
(1 row)

SELECT cast('hello world' as char(15)) = cast('hello world' as char(14));
 _col0
 -------
 false
(1 row)

SELECT cast('hello world' as varchar(15)) = cast('hello world' as varchar(14));
 _col0
 -------
 true
(1 row)

CREATE TABLE varchars(col varchar(5));

INSERT INTO into varchars values('1234');
INSERT: 1 row

INSERT INTO varchars values('123456');
Query failed: Insert query has mismatched column types:
Table: [varchar(5)], Query: [varchar(6)]
```

8.9.1 集合数据类型

随着数据变得越来越庞大和复杂，有时会以更复杂的数据类型来存储，如数组和映射等。许多 RDBMS 系统，特别是一些 NoSQL 系统，支持复杂的数据类型。Presto 支持其中的一些集合数据类型，如表 8-7 所示。它还提供对 `UNNEST` 操作的支持，参见 9.14 节。

表8-7：集合数据类型

集合数据类型	示　　例
ARRAY	ARRAY[*apples, oranges, pears*]
MAP	MAP(ARRAY[*a, b, c*]、ARRAY[1, 2, 3])
JSON	{"a":1,"b":2,"c":3}
ROW	ROW(1, 2, 3)

8.9.2 时态数据类型

表 8-8 描述了时态数据类型，即与日期和时间相关的数据类型。

表8-8：时态数据类型

类　型	描　述	示　例
DATE	包含年、月、日的日历日期	DATE '1983-10-19'
TIME	包含时、分、秒、毫秒的时间	TIME '02:56:15.123'
TIME WITH TIMEZONE	包含时、分、秒、毫秒的时间，包括一个时区	—
TIMESTAMP	一个日期和时间	—
TIMESTAMP WITH TIMEZONE	一个日期和时间，包含时区	—
INTERVAL YEAR TO MONTH	间隔时间跨度为年、月	INTERVAL '1-2' YEAR TO MONTH
INTERVAL DAY TO SECOND	间隔时间跨度为天、时、分、秒和毫秒	INTERVAL '5' DAY to SECOND

在 Presto 中，TIMESTAMP 由 Java Instant 类型表示，代表了相对于 Java epoch 前后的时间量。值被解析并以不同的格式显示，因此对终端用户来说应该是透明的。

对于不包含时区信息的类型，值会根据 Presto 会话的时区进行解析和显示；对于包含时区信息的类型，值会使用相应时区来解析并显示。

Presto 可以将字符串解析成 TIMESTAMP、TIMESTAMP WITH TIMEZONE、TIME、TIME WITH TIMEZONE 或 DATE。表 8-9 到表 8-11 描述了可解析为时间的格式。如果你想使用 ISO 8601 规范，可以使用 from_iso8601_timestamp 或 from_iso8601_date 函数。

表8-9：可解析为时间戳数据类型的字符串格式

TIMESTAMP	TIMESTAMP WITH TIMEZONE
yyyy-M-d	yyyy-M-d ZZZ
yyyy-M-d H:m	yyyy-M-d H:m ZZZ
yyyy-M-d H:m:s	yyyy-M-d H:m:s ZZZ
yyyy-M-d H:m:s.SSS	yyyy-M-d H:m:s.SSS ZZZ

表8-10：可解析为时态数据类型的字符串格式

TIME	TIMESTAMP WITH TIMEZONE
H:m	H:m ZZZ
H:m:s	H:m:s ZZZ
H:m:s.SSS	H:m:s.SSS ZZZ

表8-11：可解析为日期数据类型的字符串格式

DATE
YYYY-MM-DD

当打印 TIMESTAMP、TIMESTAMP WITH TIMEZONE、TIME、TIME WITH TIMEZONE 或 DATE 的输出

时，Presto 使用表 8-12 中的输出格式。如果你想以严格的 ISO 8601 格式输出，可以使用 `to_iso8601` 函数。

表8-12：时间输出格式

数据类型	格　　式
TIMESTAMP	yyyy-MM-dd HH:mm:ss.SSS ZZZ
TIMESTAMP WITH TIMEZONE	yyyy-MM-dd HH:mm:ss.SSS ZZZ
TIME	yyyy-MM-dd HH:mm:ss.SSS ZZZ
TIME WITH TIMEZONE	yyyy-MM-dd HH:mm:ss.SSS ZZZ
DATE	YYYY-MM-DD

1. 时区

时区添加了额外的重要信息。Presto 支持 `TIME WITH TIMEZONE`，但通常最好使用包含时区的 `DATE` 或 `TIMESTAMP` 类型，这样可以用 `DATE` 格式计算夏令时。

以下是一些时区字符串的示例：

- `America/New_York`
- `America/Los_Angeles`
- `Europe/Warsaw`
- `+08:00`
- `-10:00`

下面是一些例子：

```
SELECT TIME '02:56:15 UTC';
      _col0
 -----------------
 02:56:15.000 UTC
(1 row)

SELECT TIME '02:56:15 UTC' AT TIME ZONE 'America/Los_Angeles';
              _col0
 ---------------------------------
 18:56:15.000 America/Los_Angeles

SELECT TIME '02:56:15 UTC' AT TIME ZONE '-08:00';
        _col0
 --------------------
 18:56:15.000 -08:00
(1 row)

SELECT TIMESTAMP '1983-10-19 07:30:05.123';
          _col0
 ------------------------
 1983-10-19 07:30:05.123
(1 row)
```

```
SELECT TIMESTAMP '1983-10-19 07:30:05.123 America/New_York' AT TIME ZONE 'UTC';
            _col0
 ----------------------------
 1983-10-19 11:30:05.123 UTC
(1 row)
```

2. 时间间隔

如表 8-13 和表 8-14 所示，数据类型 `INTERVAL` 可以是 `YEAR TO MONTH` 或 `DAY TO SECOND`。

表8-13：`YEAR TO MONTH`的时间间隔

YEAR TO MONTH
INTERVAL '<years>-<months>' YEAR TO MONTH
INTERVAL '<years>' YEAR TO MONTH
INTERVAL '<years>' YEAR
INTERVAL '<months>' MONTH

表8-14：`DAY TO SECOND`的时间间隔

DAY TO SECOND
INTERVAL '<days> <time>' DAY TO SECOND
INTERVAL '<days>' DAY TO SECOND
INTERVAL '<days>' DAY
INTERVAL '<hours>' HOUR
INTERVAL '<minutes>' MINUTE
INTERVAL '<seconds>' SECOND

下面的例子展示了我们介绍过的一些行为：

```
SELECT INTERVAL '1-2' YEAR TO MONTH;
 _col0
 ------
 1-2
(1 row)

SELECT INTERVAL '4' MONTH;
 _col0
 -------
 0-4
(1 row)

SELECT INTERVAL '4-1' DAY TO SECOND;
Query xyz failed: Invalid INTERVAL DAY TO SECOND value: 4-1

SELECT INTERVAL '4' DAY TO SECOND;
     _col0
 ----------------
 4 00:00:00.000
(1 row)
```

```
SELECT INTERVAL '4 01:03:05.44' DAY TO SECOND;
     _col0
 ----------------
 4 01:03:05.440
(1 row)

SELECT INTERVAL '05.44' SECOND;
     _col0
 ----------------
 0 00:00:05.440
(1 row)
```

8.9.3 类型转换

有时需要将值或字符串显式地转换为不同的数据类型，这就是所谓的**类型转换**（type casting），它由 `CAST` 函数执行：

```
CAST(value AS type)
```

如果需要将日期与字符串进行比较：

```
SELECT *
FROM hive.web.page_views
WHERE view_date > '2019-01-01';
Query failed: line 1:42: '>' cannot be applied to date, varchar(10)
```

这个查询会失败，因为 Presto 没有大于（>）比较运算符，所以不知道如何比较日期和字符串。但是，它的比较函数能够比较两个日期。因此，我们需要使用 `CAST` 函数来强制转换其中的一个类型。在这个例子中，我们将字符串转换为日期：

```
SELECT *
FROM hive.web.page_views
WHERE view_date > CAST('2019-01-01' as DATE);
        view_time        | user_id | page_url | view_data  | country
 ------------------------+---------+----------+------------+---------
 2019-01-26 20:40:15.477 |       2 | http://  | 2019-01-26 | US
 2019-01-26 20:41:01.243 |       3 | http://  | 2019-01-26 | US
...
```

Presto 提供了另一个转换函数——`try_cast`。它尝试执行类型转换，但与失败后会报错的 `CAST` 不同，`try_cast` 会返回 `NULL` 值，这在不需要处理错误时很有用：

```
try_cast(value AS type)
```

举个例子，让我们把字符字面值强制转换为数字类型：

```
SELECT cast('1' AS integer);
 _col0
 -------
     1
(1 row)
```

```
SELECT cast('a' as integer);
Query failed: Cannot cast 'a' to INT

SELECT try_cast('a' as integer);
 _col0
 ------
 NULL
(1 row)
```

8.10 SELECT语句基础

SELECT 语句非常重要，它允许你以表的格式从一个或多个表中返回数据，也可以仅仅返回一行或一个值。

Presto 的 SELECT 查询更加复杂，因为它可以包含来自不同 catalog 和 schema 的表，而这些是完全不同的数据源。你在 7.6 节中已经了解了这一点。

下面我们将深入细节，全面了解其能力。让我们从语法概述开始：

```
[ WITH with_query [, ...] ]
SELECT [ ALL | DISTINCT ] select_expr [, ...]
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY [ ALL | DISTINCT ] grouping_element [, ...] ]
[ HAVING condition]
[ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
[ ORDER BY expression [ ASC | DESC ] [, ...] ]
[ LIMIT [ count | ALL ] ]
```

select_expr 表示查询以表列、派生表列、常量或一般表达式的形式返回的数据，其形式为零行、一行或多行。一般表达式可以包括函数、操作符、列和常量。你可以只用 SELECT select_expr 来运行一个查询，它的作用仅限于测试。

```
SELECT 1, 1+1, upper('lower');
 _col0 | _col1 | _col2
-------+-------+------
     1 |     2 | LOWER
(1 row)
```

SELECT select_expr [, ...] FROM from_item 是最基本的查询形式。它允许你从一个底层表中取出所有的数据，或者只取出一些列。它还允许你对底层数据使用计算表达式。

假设我们有两个表（也称为两组关系）nation 和 customer。这些例子取自我们在 6.3 节中讨论过的 TPC-H 数据集。简洁起见，示例表只截取了几行和几列显示。在本章的多个 select 查询例子中，我们都将使用这些数据。

你可以从 sf1 schema 中的 nation 表返回所选择列的数据：

```
SELECT nationkey, name, regionkey
FROM tpch.sf1.nation;
 nationkey |      name      | regionkey
-----------+----------------+-----------
         0 | ALGERIA        |         0
         1 | ARGENTINA      |         1
         2 | BRAZIL         |         1
         3 | CANADA         |         1
         4 | EGYPT          |         4
         5 | ETHIOPIA       |         0
...
```

或是从 customer 表获得一些数据：

```
SELECT custkey, nationkey, phone, acctbal, mktsegment
FROM tpch.tiny.customer;
 custkey | nationkey |      phone      | acctbal | mktsegment
---------+-----------+-----------------+---------+------------
     751 |         0 | 10-658-550-2257 | 2130.98 | FURNITURE
     752 |         8 | 18-924-993-6038 | 8363.66 | MACHINERY
     753 |        17 | 27-817-126-3646 | 8114.44 | HOUSEHOLD
     754 |         0 | 10-646-595-5871 | -566.86 | BUILDING
     755 |        16 | 26-395-247-2207 | 7631.94 | HOUSEHOLD
     756 |        14 | 24-267-298-7503 | 8116.99 | AUTOMOBILE
     757 |         3 | 13-704-408-2991 | 9334.82 | AUTOMOBILE
     758 |        17 | 27-175-799-9168 | 6352.14 | HOUSEHOLD
...
```

除了单纯地返回所选择的数据，我们还可以用函数进行数据转换并返回结果：

```
SELECT acctbal, round(acctbal) FROM tpch.sf1.customer;
 acctbal | _col1
---------+--------
 7470.96 | 7471.0
 8462.17 | 8462.0
 2757.45 | 2757.0
 -588.38 | -588.0
 9091.82 | 9092.0
 3288.42 | 3288.0
 2514.15 | 2514.0
 2259.38 | 2259.0
  -716.1 | -716.0
 7462.99 | 7463.0
(10 rows)
```

8.11 WHERE子句

在 SELECT 查询中可以使用 WHERE 子句指定过滤条件。它由一个结果为 TRUE、FALSE 或 UNKNOWN 的条件所组成。在执行查询的过程中，该条件对每一行求值。如果结果不等于 TRUE，该行将被跳过且不会出现在结果集中；否则，该行将作为结果集的一部分返回给用户或用于进一步处理。

WHERE 子句的条件由一个或多个布尔表达式组成，这些表达式由 AND 和 OR 相连。

```
SELECT custkey, acctbal
FROM tpch.sf1.customer WHERE acctbal < 0;
 custkey | acctbal
---------+---------
   75016 | -735.89
   75027 | -399.78
   75028 | -222.92
   75034 | -679.38
   75037 | -660.07
...

SELECT custkey, acctbal FROM tpch.sf1.customer
WHERE acctbal > 0 AND acctbal < 500;
 custkey | acctbal
---------+---------
   75011 |  165.71
   75012 |   41.65
   75021 |   176.2
   75022 |  348.24
   75026 |   78.64
   75084 |   18.68
   75107 |  314.88
...
```

WHERE 子句的条件很重要，因为它可用于多种查询优化。在 4.8 节中，你可以了解更多关于查询优化的内容。当查询多个表时，你可以通过 WHERE 子句中的条件将它们连接起来。Presto 使用这些信息来决定高效的查询执行计划。

8.12 GROUP BY和HAVING子句

GROUP BY 和 HAVING 子句在分析型查询中非常常用。GROUP BY 用于将具有相同值的行合并成一行。

```
SELECT mktsegment
FROM tpch.sf1.customer
GROUP BY mktsegment;
 mktsegment
 -----------
 MACHINERY
 AUTOMOBILE
 HOUSEHOLD
 BUILDING
 FURNITURE
(5 rows)
```

对于 Presto 中的分析型查询，GROUP BY 经常与聚合函数相结合。这些函数将每一组的数据聚合成一个值。下面的查询计算按细分市场分组后每一组内所有客户的全部账户余额。

```
SELECT mktsegment, round(sum(acctbal) / 1000000, 3) AS acctbal_millions
FROM tpch.sf1.customer
GROUP BY mktsegment;
 mktsegment | acctbal_millions
------------+------------------
 MACHINERY  |          134.439
 AUTOMOBILE |          133.867
 BUILDING   |          135.889
 FURNITURE  |          134.259
 HOUSEHOLD  |          135.873
```

即使不使用 GROUP BY 子句，也可以使用聚合函数。在这种情况下，整个表作为聚合函数的输入，因此，我们可以计算出全部账户的总余额：

```
SELECT round(sum(acctbal) / 1000000, 3) AS acctbal_millions
FROM tpch.sf1.customer;
 acctbal_millions
------------------
          674.327
```

HAVING 子句类似于 WHERE 子句。它对每一条记录进行求值，只有当结果为 TRUE 时才计入该记录。HAVING 子句在 GROUP BY 之后求值，并作用于分组后的行；而 WHERE 子句是在 GROUP BY 之前求值的，并作用于单个行。

下面是完整的查询：

```
SELECT mktsegment,
      round(sum(acctbal), 1) AS acctbal_per_mktsegment
FROM tpch.tiny.customer
GROUP BY mktsegment
HAVING round(sum(acctbal), 1) > 5283.0;
 mktsegment | acctbal_per_mktsegment
------------+------------------------
 BUILDING   |              1444587.8
 HOUSEHOLD  |              1279340.7
 AUTOMOBILE |              1395695.7
 FURNITURE  |              1265282.8
 MACHINERY  |              1296958.6
(5 rows)
```

下面是在已分组的数据上进行条件过滤：

```
SELECT mktsegment,
       round(sum(acctbal), 1) AS acctbal_per_mktsegment
FROM tpch.tiny.customer
GROUP BY mktsegment
HAVING round(sum(acctbal), 1) > 1300000;
 mktsegment | acctbal_per_mktsegment
------------+------------------------
 AUTOMOBILE |              1395695.7
 BUILDING   |              1444587.8
(2 rows)
```

8.13 ORDER BY子句和LIMIT子句

ORDER BY 子句含有用于对结果进行排序的表达式。该子句可以包含多个表达式，是从左到右进行计算的。通常，当左边的表达式计算结果中出现重复值时，会使用右边的表达式来确定顺序。表达式可以指定排序顺序为升序（例如 *A*~*Z*、1~100）或降序（例如 *Z*~*A*、100~1）。

LIMIT 子句用于限制返回的行数，并且可以与 ORDER BY 子句结合使用来查找有序集的前 *N* 个结果：

```
SELECT mktsegment,
       round(sum(acctbal), 2) AS acctbal_per_mktsegment
FROM tpch.sf1.customer
GROUP BY mktsegment
HAVING sum(acctbal) > 0
ORDER BY acctbal_per_mktsegment DESC
LIMIT 1;
 mktsegment | acctbal_per_mktsegment
------------+------------------------
 MACHINERY  |               19851.2
(1 row)
```

通常情况下，Presto 能够将 ORDER BY 和 LIMIT 作为一个组合步骤来优化执行，而不是分开执行。

LIMIT 可以不使用 ORDER BY 子句，但大多数情况下它们是一起使用的。原因是 SQL 标准（继而也是 Presto 的标准）并不保证结果的顺序。这就意味着，在没有 ORDER BY 子句的情况下使用 LIMIT，多次运行同一个查询可能返回不同的、非确定性的结果。这种情况在像 Presto 这样的分布式系统中更加明显。

8.14 JOIN语句

SQL 可以让你借助 JOIN 语句来组合不同表的数据。Presto 支持 SQL 标准的 JOIN 语句，如 INNER JOIN、LEFT OUTER JOIN、RIGHT OUTER JOIN、FULL OUTER JOIN、FULL OUTER JOIN 和 CROSS JOIN 等。对 JOIN 语句的完整介绍超出了本书的范围，但在其他许多书中有所涉及。

让我们来看几个例子，并探讨与 Presto 相关的具体细节：

```
SELECT custkey, mktsegment, nation.name AS nation
FROM tpch.tiny.nation JOIN tpch.tiny.customer
ON nation.nationkey = customer.nationkey;
 custkey | mktsegment |   nation
---------+------------+-----------
     108 | BUILDING   | ETHIOPIA
     101 | MACHINERY  | BRAZIL
     106 | MACHINERY  | ARGENTINA
(3 rows)
```

Presto 可以使用隐式 Cross Join：用一个以逗号分隔表名的列表定义 Join 在一起的表，并用 WHERE 子句来定义 Join 条件。

```
SELECT custkey, mktsegment, nation.name AS nation
FROM tpch.tiny.nation, tpch.tiny.customer
WHERE nation.nationkey = customer.nationkey;
 custkey | mktsegment |   name
---------+------------+-----------
     108 | BUILDING   | ETHIOPIA
     106 | MACHINERY  | ARGENTINA
     101 | MACHINERY  | BRAZIL
```

Join 是查询处理中开销最大的操作之一。当一个查询中存在多个 Join 时，可以以不同的顺序处理这些 Join。TPCH 基准中的 Q09 查询就是一个很好的复杂查询示例：

```
SELECT
  nation,
  o_year,
  sum(amount) AS sum_profit
FROM (
       SELECT
         N.name AS nation,
         extract(YEAR FROM o.orderdate)AS o_year,
         l.extendedprice * (1 - l.discount) - ps.supplycost * l.quantity AS amount
       FROM
         part AS p,
         supplier AS s,
         lineitem AS l,
         partsupp AS ps,
         orders AS o,
         nation AS n
       WHERE
         s.suppkey = l.suppkey
         AND ps.suppkey = l.suppkey
         AND ps.partkey = l.partkey
         AND p.partkey = l.partkey
         AND o.orderkey = l.orderkey
         AND s.nationkey = n.nationkey
         AND p.name LIKE '%green%'
     ) AS profit
GROUP BY
  nation,
  o_year
ORDER BY
  nation,
  o_year DESC;
```

8.15 UNION、INTERSECT和EXCEPT子句

UNION、INTERSECT 和 EXCEPT 在 SQL 中叫作**集合操作**。它们用于将多个 SQL 语句中的数据组合成一个结果。

虽然你可以使用 Join 和条件来获得相同的语义，但通常使用集合操作更容易。在 Presto 中执行集合操作比在其他等价的 SQL 中执行它们更高效。

在学习集合操作的语义时，从基本的整数开始会比较容易。首先是 `UNION`，它可以合并所有的值并去除重复的值：

```
SELECT * FROM (VALUES 1, 2)
UNION
SELECT * FROM (VALUES 2, 3);
 _col0
 ------
     2
     1
     3
(3 rows)
```

`UNION ALL` 将保留重复值：

```
SELECT * FROM (VALUES 1, 2)
UNION ALL
SELECT * FROM (VALUES 2, 3);
 _col0
  -----
     1
     2
     2
     3
(4 rows)
```

`INTERSECT` 返回两个查询中都有的元素作为结果集：

```
SELECT * FROM (VALUES 1, 2)
INTERSECT
SELECT * FROM (VALUES 2, 3);
 _col0
 ------
     2
(1 row)
```

`EXCEPT` 返回第一个查询中去除第二个查询中的元素后剩下的元素：

```
SELECT * FROM (VALUES 1, 2)
EXCEPT
SELECT * FROM (VALUES 2, 3);
 _col0
 ------
     1
(1 row)
```

每个集合操作符都支持使用可选的修饰符：`DISTINCT` 或 `ALL`。`DISTINCT` 关键字是默认的，无须指定；`ALL` 关键字表示保留重复值。目前，`INTERSECT` 和 `EXCEPT` 运算符不支持 `ALL`。

8.16 分组操作

你已经学习了基本的 GROUP BY 和聚合操作。Presto 还支持 SQL 标准中的高级分组操作。使用 GROUPING SETS、CUBE 和 ROLLUP，用户可以在单个查询中对多个集合进行聚合操作。

分组集合允许你在同一个查询中按照多个列的列表进行分组。假设我们想按照 (state, city, street)、(state, city) 和 (state) 进行分组。如果没有分组集合，则你必须针对每种分组运行单独的查询，然后再合并结果；而有了分组集合，Presto 可以根据每个集合计算分组，结果 schema 是所有集合中列的联合。对于不属于组的列，将添加 NULL。

ROLLUP 和 CUBE 可以用 GROUPING SETS 来表示，这是它的一种简写。ROLLUP 用于生成基于层次结构的分组集合，例如 ROLLUP(a, b, c) 生成分组集合 (a, b, c)、(a, b)、(a)、()；而 CUBE 则操作生成所有可能的分组组合，例如 CUBE(a, b, c) 生成分组集合 (a, b, c)、(a, b)、(a, c)、(b, c)、(a)、(b)、(c)、()。

假设你想计算每个细分市场的账户余额总额，同时计算所有细分市场的账户总余额：

```
SELECT mktsegment,
  round(sum(acctbal), 2) AS total_acctbal,
  GROUPING(mktsegment) AS id
FROM tpch.tiny.customer
GROUP BY ROLLUP (mktsegment)
ORDER BY id, total_acctbal;
 mktsegment | total_acctbal | id
------------+---------------+----
 FURNITURE  |     1265282.8 |  0
 HOUSEHOLD  |    1279340.66 |  0
 MACHINERY  |    1296958.61 |  0
 AUTOMOBILE |    1395695.72 |  0
 BUILDING   |     1444587.8 |  0
 NULL       |    6681865.59 |  1
(6 rows)
```

借助 ROLLUP 可以计算不同分组的聚合。在这个例子中，前五行代表每个细分市场的账户余额总和，最后一行代表所有账户余额的总和。因为这一分组中没有 mktsegment 列，所以填充 NULL 值。GROUPING 函数用于标识哪些行属于哪个组。

如果没有 ROLLUP，则你必须运行两个单独的查询，然后把它们组合在一起。在这个例子中，我们可以使用 UNION，它可以帮助你从概念上理解 ROLLUP 的作用。

```
SELECT mktsegment,
       round(sum(acctbal), 2) AS total_acctbal,
       0 AS id
FROM tpch.tiny.customer
GROUP BY mktsegment
UNION
SELECT NULL, round(sum(acctbal), 2), 1
```

```
FROM tpch.tiny.customer
ORDER BY id, total_acctbal;
mktsegment | total_acctbal | id
------------+---------------+----
 FURNITURE  |     1265282.8 |  0
 HOUSEHOLD  |    1279340.66 |  0
 MACHINERY  |    1296958.61 |  0
 AUTOMOBILE |    1395695.72 |  0
 BUILDING   |     1444587.8 |  0
 NULL       |    6681865.59 |  1
(6 rows)
```

8.17 WITH子句

WITH 子句用于在单个查询中定义一个内联视图。这通常可以使查询更可读，因为查询可能需要多次包含相同的嵌套查询。

在下面这个查询中，我们查找有哪些细分市场的账户总余额大于细分市场的平均数：

```
SELECT mktsegment,
       total_per_mktsegment,
       average
FROM
  (
    SELECT mktsegment,
       round(sum(acctbal)) AS total_per_mktsegment
    FROM tpch.tiny.customer
    GROUP BY 1
  ),
  (
    SELECT round(avg(total_per_mktsegment)) AS average
    FROM
      (
        SELECT mktsegment,
           sum(acctbal) AS total_per_mktsegment
        FROM tpch.tiny.customer
        GROUP BY 1
      )
  )
WHERE total_per_mktsegment > average;
 mktsegment | total_per_mktsegment |  average
------------+----------------------+-----------
 BUILDING   |            1444588.0 | 1336373.0
 AUTOMOBILE |            1395696.0 | 1336373.0
(2 rows)
```

可以看到，这个查询有点复杂。使用 WITH 子句，可以将其简化成：

```
WITH
total AS (
  SELECT mktsegment,
```

```
    round(sum(acctbal)) AS total_per_mktsegment
  FROM tpch.tiny.customer
  GROUP BY 1
),
average AS (
  SELECT round(avg(total_per_mktsegment)) AS average
  FROM total
)
SELECT mktsegment,
  total_per_mktsegment,
  average
FROM total,
  average
WHERE total_per_mktsegment > average;
mktsegment | total_per_mktsegment |  average
-----------+----------------------+-----------
 AUTOMOBILE |            1395696.0 | 1336373.0
 BUILDING   |            1444588.0 | 1336373.0
(2 rows)
```

在这个例子中，第二个内联视图引用了第一个视图。可以看到 WITH 内联视图被执行了两次。目前，Presto 并不会将结果物化以便在多个执行中共享。事实上，这需要根据查询的复杂程度来进行基于代价的决策，因为多次执行一个查询可能比先保存再检索结果更高效。

8.18 子查询

Presto 支持许多常见的子查询用法。**子查询**是一个表达式，它可以作为更高级别的表达式的输入。在 SQL 中，子查询可以分为三类：

- 标量子查询
- ANY/SOME
- ALL

每个类别都有两种类型：非关联子查询和关联子查询。关联子查询指引用子查询之外的其他列的子查询。

8.18.1 标量子查询

标量子查询指返回单个值（一行一列）的子查询：

```
SELECT regionkey, name
FROM tpch.tiny.nation
WHERE regionkey =
  (SELECT regionkey FROM tpch.tiny.region WHERE name = 'AMERICA');
 regionkey |     name
-----------+---------------
         1 | ARGENTINA
         1 | BRAZIL
```

```
          1 | CANADA
          1 | PERU
          1 | UNITED STATES
(5 rows)
```

在这个标量子查询示例中，子查询的结果是 1，WHERE 条件实质上变成了 regionkey = 1，并且对每行进行计算。逻辑上来说，对 nation 表中的每一行都要进行子查询的计算，例如，对 100 行计算 100 次。然而，Presto 足够智能：它只对子查询进行一次计算，之后都使用静态值。

8.18.2 EXISTS子查询

EXISTS 子查询表示如果有任何记录则结果为 true。这些查询通常作为关联子查询使用。虽然对于 EXISTS 来说，非关联子查询也是可以的，但这并不实用，因为任何返回单行的东西都会计算为 true：

```
SELECT name
FROM tpch.tiny.nation
WHERE regionkey IN (SELECT regionkey FROM tpch.tiny.region)
```

EXISTS 子查询的另一种常见形式是 NOT EXISTS。不过这只是对 EXISTS 子查询的结果取否。

8.18.3 集合比较子查询

ANY 子查询的形式为表达式运算符量化符（子查询）。有效的运算符的值是 <、>、<=、>=、>=、= 或 <>。也可以使用 SOME 来代替 ANY。这种类型的查询最常见的另一个形式是表达式 IN 子查询，它等价于表达式 = ANY 子查询。

```
SELECT name
FROM nation
WHERE regionkey = ANY (SELECT regionkey FROM region)
```

这个查询等价于下面的查询，其中 IN 是个简写：

```
SELECT name
FROM nation
WHERE regionkey IN (SELECT regionkey FROM region)
```

该子查询必须恰好返回一个列。Presto 目前不支持行表达式子查询，即不支持多列比较。从语义上来说，对于外部查询的给定行，计算子查询，并将表达式与子查询的每个结果行比较。如果这些比较中至少有一个比较结果为 TRUE，则 ANY 子查询条件的结果为 TRUE；如果没有一个比较结果为 TRUE，则结果为 FALSE。对外部查询的每一行都会重复进行这样的计算。

注意一些细微的差别。如果表达式是 NULL，那么 IN 表达式的结果就是 NULL。此外，如果

没有比较结果为 TRUE，但子查询中存在一个 NULL 值，那么 IN 表达式的结果是 NULL。在大多数情况下这不会被注意到，因为结果无论是 FALSE 还是 NULL 都会导致过滤该行。但是，如果这个 IN 表达式是作为对 NULL 值敏感的表达式的输入（例如，用 NOT 修饰），这个差异就会表现出来。

ALL 子查询的工作原理与 ANY 类似。对于外部查询中的某一行，计算子查询，并将表达式与子查询的每个结果行比较。如果所有的比较结果都为 TRUE，则 ALL 的结果为 TRUE；如果至少有一个比较结果为 FALSE，则 ALL 的结果为 FALSE。

与 ANY 一样，ALL 也有一些初看起来并不明显的细微差别。当子查询为空且没有返回任何行时，ALL 就会计算为 TRUE；如果没有一个比较结果返回 FALSE 且至少有一个比较结果返回 NULL，那么 ALL 的结果就是 NULL。ALL 最常见的另一个形式是 <> ALL，等价于 NOT IN。

8.19　从表中删除数据

DELETE 语句可以删除表中的数据行。该语句提供了一个可选的 WHERE 子句来指定删除哪些行。若不使用 WHERE 子句，则表中所有数据都会被删除：

```
DELETE FROM table_name [ WHERE condition ]
```

有些连接器不支持删除或有限支持删除。例如，Kafka 连接器不支持删除；Hive 连接器只有在 WHERE 子句指定了可用于删除整个分区的分区键时，才支持删除：

```
DELETE FROM hive.web.page_views
WHERE view_date = DATE '2019-01-14' AND country = 'US'
```

8.20　小结

对你在 Presto 中可以用 SQL 做到的事感到兴奋了吗？有了本章的知识，你已经可以对 Presto 中的任何数据进行复杂的查询，并完成一些相当复杂的分析。

当然，还有更多内容等着你。请继续阅读第 9 章，了解 Presto 的函数、运算符和其他功能。

第 9 章

高级SQL特性

如第 8 章所述，利用 SQL 语句的强大功能，可以实现很多东西，但你对 Presto 中的查询还只是略知皮毛。本章将涉及更多的高级特性，如函数、运算符等。

9.1　函数和运算符介绍

到目前为止，你已经了解了一些基本的知识，包括 catalog、schema、表、数据类型和各种 SQL 语句等。这些知识在 Presto 中查询一个或多个 catalog 的一个或多个表的数据时非常有用。在那些例子中，我们主要借助表中不同属性（或列）中的数据编写查询。

而 SQL 函数和运算符的存在则是为了实现更复杂、全面的 SQL 查询。本章将重点介绍 Presto 所支持的函数和运算符，并提供使用示例。

SQL 中的函数和运算符在内部等价。函数一般使用的语法形式是 `function_name(function_arg1, ...)`，而运算符使用的是不同语法，类似于编程语言和数学中的运算符，运算符是常用函数的语法缩写和改进形式。运算符与函数相互等价的一个例子是 `||` 运算符和 `concat()` 函数，二者都是用来连接字符串的。

SQL 和 Presto 中的运算符一般有以下两种类型。

双目运算符

双目运算符取两个操作数作为输入，并产生一个单值结果。运算符本身定义了操作数的数据类型必须是什么，结果的数据类型是什么。双目运算符的格式是**操作数运算符操作数**。

单目运算符

单目运算符接收单个操作数的输入，并产生一个单值结果。与双目运算符一样，运算符指定了操作数的数据类型必须是什么，结果的数据类型必须是什么。单目运算符写作**运算符操作数**。

对于双目运算符和单目运算符，输入的操作数可以是一个运算符树运算的结果。例如，在 2×2×2 中，第一个运算符 2×2 的结果被输入第二个乘法运算符中。

在本章的以下几节中，你将详细了解 Presto 所支持的众多函数和运算符。

9.2 标量函数和运算符

抽象地讲，SQL 中的标量函数将一个或多个单值作为输入参数，根据输入的参数执行一个操作，然后产生一个单值。例如 `power(x, p)` 函数返回 *x* 的 *p* 次幂。

```
SELECT power(2, 3);
 _col0
-------
   8.0
(1 row)
```

当然，你可以用乘法运算符来实现同样的目的，在本例中，2×2×2 也会产生值 8。但是，使用函数可以将逻辑封装起来，使其更容易在 SQL 中重复使用。此外，还能得到其他收益，比如减少出错的可能和优化函数的执行。

只要在语义正确，标量函数可以用在 SQL 语句中任何可以使用表达式的地方。例如，你可以写一个 SQL 查询 `SELECT * FROM page_views WHERE power(2, 3)`。它可以通过语法检查，但在语义分析时会失败，因为 `power` 函数的返回类型是 `DOUBLE` 而不是所需的 `BOOLEAN` 类型。可以将 SQL 查询写成 `SELECT * FROM page_views WHERE power(2, 3) = 8`，虽然它可能没有实际用处，但语义上是正确的。

Presto 包含了一组可以立即使用的内置函数和运算符。在本节中，你了解了常见用法并着重学习了有趣的部分。我们并没有列举每一个函数和运算符，因为本章无意作为全面的参考资料。学习了一些基础知识后，你可以参考 Presto 文档来了解更多。



Presto 还支持**用户自定义函数**（user-defined function，UDF），它允许你使用 Java 编写自己的函数实现，并在 Presto 中部署这些函数，从而在 SQL 查询中执行它，但这超出了本书的范围。

9.3 布尔运算符

布尔运算符是双目运算符，在 SQL 中用于比较两个操作数的值，从而产生 TRUE、FALSE 或 NULL 的布尔型结果。这些运算符（见表 9-1）最常用于 WHERE、HAVING 或 ON 等条件子句。可以在查询中任何能够使用表达式的地方使用它们。

表9-1：布尔运算符

运 算 符	描 述
<	小于
<=	小于等于
>	大于
>=	大于等于
=	相等
<>	不相等
!=	不相等



语法 != 不属于标准 SQL，但在各种编程语言中普遍使用。许多流行的数据库中也有实现，因此在 Presto 中也提供了这个语法以方便使用。

下面是一些布尔运算符的使用示例。

2 月某周中飞离波士顿最佳日子是哪一天？

```
SELECT dayofweek, avg(depdelayminutes) AS delay
FROM flights_orc
WHERE month = 2 AND origincityname LIKE '%Boston%'
GROUP BY dayofweek
ORDER BY dayofweek;

 dayofweek |       delay
-----------+--------------------
         1 | 10.613156692553677
         2 |   9.97405624214174
         3 |  9.548045977011494
         4 | 11.822725778003647
         5 | 15.875475113122173
         6 | 11.184173669467787
         7 | 10.788121285791464
(7 rows)
```

从 2010 年到 2014 年，每年每个航空公司的平均延误时间是多少？

```
SELECT avg(arrdelayminutes) AS avg_arrival_delay, carrier
FROM flights_orc
```

```
WHERE year > 2010 AND year < 2014
GROUP BY carrier, year;

 avg_arrival_delay  | carrier
--------------------+---------
 11.755326255888736 | 9E
 12.557365851045104 | AA
  13.39056266711295 | AA
 13.302276406082575 | AA
 6.4657695873247745 | AS
  7.048865559750841 | AS
  6.907012760530203 | AS
 17.008730526574663 | B6
  13.28933909176506 | B6
 16.242635221309246 | B6
...
```

9.4 逻辑运算符

在 SQL 和 Presto 中还有三个**逻辑运算符**：AND、OR 和 NOT。运算符 AND 和 OR 是**双目运算符**，因为它们接受两个参数作为输入。NOT 是一个单目运算符，它只接受一个参数作为输入。

这些逻辑运算符接收 BOOLEAN 类型的输入数据，返回单个布尔变量 TRUE、FALSE 或 NULL (UNKNOWN)。通常，这些运算符组合使用形成条件子句，就像布尔运算符那样。

这三个运算符的概念与编程语言相似，但由于 NULL 值的存在，语义有所不同。例如，NULL AND NULL 不等于 TRUE，而是等于 NULL。如果你把 NULL 看作值的缺失，这就很容易理解了。表 9-2 展示了运算符如何处理这三个值。

表9-2：AND和OR的逻辑运算结果

x	*y*	*x* AND *y*	*x* OR *y*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	TRUE	NULL	TRUE
NULL	FALSE	FALSE	NULL
NULL	NULL	NULL	NULL

而对于 NOT 运算符，只要记住 NOT NULL 结果是 NULL，而不是 TRUE 或 FALSE。

9.5 用BETWEEN语句选择范围

BETWEEN **语句**可以看作双目运算符的特殊情况，它定义了一个范围。它实际上等价于由 AND 连接起来的两个比较项，所选字段的数据类型和两个范围值必须相同。NOT BETWEEN 是 BETWEEN 的否定。从结果也可以看出，下面这两个查询是等价的：

```
SELECT count(*) FROM flights_orc WHERE year BETWEEN 2010 AND 2012;
  _col0
----------
 18632160
(1 row)

SELECT count(*) FROM flights_orc WHERE year >= 2010 AND year <= 2012;
  _col0
----------
 18632160
(1 row)
```

9.6 用IS（NOT）NULL检测值的存在

IS NULL 语句允许你检测值是否存在。它可以被认为是一种特殊类型的单目运算符。IS NOT NULL 是其否定形式。你可能想统计一些行，但不想统计没有值的行。这是因为这些数据可能不完整或没有意义。

例如，要计算每年飞机的平均延误时间，必须确保只计算实际发生的航班。这一点体现在 airtime 要有一个值，下面这个查询就考虑到了这一点：

```
SELECT avg(DepDelayMinutes) AS delay, year
FROM flights_orc
WHERE airtime IS NOT NULL and year >= 2015
GROUP BY year
ORDER BY year desc;
       delay        | year
--------------------+------
 12.041834908176538 | 2019
 13.178923805354275 | 2018
 12.373612267166829 | 2017
  10.51195619395339 | 2016
 11.047527214544516 | 2015
(5 rows)
```

9.7 数学函数和运算符

使用数学函数和运算符开启了广泛的用例，并对其中很多用例至关重要。表 9-3 列出了数学运算符，表 9-4 列出了数学函数。

表9-3：数学运算符

运 算 符	描 述	示 例
+	加	SELECT 1+1
-	减	SELECT 2-1
*	乘	SELECT 2*3
/	除	SELECT 9/2
%	取模	SELECT 6 % 5

表9-4：常用数学函数

函 数	返回类型	描 述	示 例
abs(*x*)	同输入	*x* 的绝对值	SELECT abs(-1)
cbrt(*x*)	DOUBLE	*x* 的立方根	SELECT cbrt(9)
ceiling(*x*)	同输入	将 *x* 向上取最接近的整数	SELECT ceiling(4.2)
degrees(*x*)	DOUBLE	将 *x* 角从弧度转换为角度	SELECT degrees(1.047)
exp(*x*)	DOUBLE	e（自然对数的底）的 *x* 次方	SELECT exp(1)
floor(*x*)	同输入	将 *x* 向下取最接近的整数	SELECT floor(4.2)
ln(*x*)	DOUBLE	*x* 的自然对数	SELECT ln(exp(1))
log(*b*, *x*)	DOUBLE	*x* 的以 b 为底的对数	SELECT log(2, 64)
log2(*x*)	DOUBLE	*x* 的以 2 为底的对数	SELECT log2(64)
log10(*x*)	DOUBLE	*x* 的以 10 为底的对数	SELECT log10(140)
mod(*n*, *m*)	同输入	取模，相当于 *n* % *m*	SELECT mod(3, 2)
power(*x*, *p*)	DOUBLE	*x* 的 *p* 次方	SELECT pow(2, 6)
radians(*x*)	DOUBLE	将 *x* 从角度转换为弧度	SELECT radians(60)
round(*x*)	同输入	将 *x* 四舍五入至最接近的整数	SELECT round(pi())
round(*x*, *d*)	同输入	将 *x* 四舍五入到小数点后的 *d* 位	SELECT round(pi(), 2)
sqrt(*x*)	DOUBLE	*x* 的平方根	SELECT sqrt(64)
truncate(*x*)	DOUBLE	通过截断小数点后的数字将 *x* 舍为整数	SELECT truncate(e())

9.8 三角函数

Presto 提供了一组**三角函数**，其参数类型为弧度，这组函数的返回数据类型都是 DOUBLE。

如果你想在弧度和角度之间进行转换，Presto 提供了 radians(x) 和 degrees(x) 这两个转换函数。表 9-5 列出了可用的三角函数。

表9-5：三角函数

函 数	描 述
cos(*x*)	*x* 的余弦
acos(*x*)	*x* 的反余弦
cosh(*x*)	*x* 的双曲余弦

（续）

函　数	描　述
sin(*x*)	*x* 的正弦
asin(*x*)	*x* 的反正弦
tan(*x*)	*x* 的正切
atan(*x*)	*x* 的反正切
atan2(*y*, *x*)	*y*/*x* 的反正切
tanh(*x*)	*x* 的双曲正切

9.9　常数和随机函数

Presto 提供了一组函数，它们返回**数学常数**、概念值或随机值，如表 9-6 所示。

表9-6：数学常数和函数

函　数	返回类型	描　述	示　例
e()	DOUBLE	欧拉数	2.718281828459045
pi()	DOUBLE	圆周率	3.141592653589793
infinity()	DOUBLE	用于表示无穷大的 Presto 常量	Infinity
nan()	DOUBLE	用于表示不是一个数字的 Presto 常量	NaN
random()	DOUBLE	大于等于 0.0 且小于 1.0 的 DOUBLE 数值	SELECT random()
random(*n*)	同输入	大于等于 0.0 并小于 *n* 的 DOUBLE 数值	SELECT random(100)

9.10　字符串函数和运算符

字符串操作是另一种常见的用例，对此 Presto 提供了丰富的支持。|| 操作符用于将字符串拼接起来：

```
SELECT 'Emily' || ' Grace';
    _col0
-------------
 Emily Grace
(1 row)
```

Presto 提供了几个有用的字符串函数，如表 9-7 所示。

表9-7：字符串函数

函　数	返回类型	描　述	示　例
chr(*n*)	VARCHAR	返回 Unicode 码点 *n* 代表的单个字符	SELECT chr(65)
codepoint(*string*)	INTEGER	返回一个字符串的 Unicode	SELECT codepoint(*A*)
concat(*string1*, ..., *stringN*)	VARCHAR	连接两个字符串	SELECT concat(*Emily*, ' , '*Grace*);

（续）

函　　数	返回类型	描　　述	示　　例
length(*string*)	BIGINT	返回一个字符串的长度	SELECT length(*saippuakivikauppias*)
lower(*string*)	VARCHAR	将字符串转换为小写	SELECT lower(*UPPER*);
lpad(*string*, *size*, *padstring*)	VARCHAR	在字符串左边填充 *size* 个 *padstring*。如果 *size* 小于字符串的实际长度，则截断该字符串	SELECT lpad(*A*, 4, ' ')
ltrim(*string*)	VARCHAR	去除头部空格	SELECT ltrim(lpad(*A*, 4, ' '))
replace(*string*, *search*, *replace*)	VARCHAR	将 *string* 中的 *search* 字符串替换为 *replace* 字符串	SELECT replace(*555.555.5555*, ., -)
reverse(*string*)	VARCHAR	反转字符串	SELECT reverse(*saippuakivikauppias*)
rpad(*string*, *size*, *padstring*)	VARCHAR	在字符串右边填充 *size* 个 *padstring*。如果 *size* 小于字符串的实际长度，则截断该字符串	SELECT rpad(*A*, 4, #)
rtrim(*string*)	VARCHAR	去除尾部空格	SELECT rtrim(rpad(*A*, 4, ' '))
split(*string*, *delimiter*)	ARRAY(VARCHAR)	对 *string* 以 *delimiter* 进行分割，返回一个数组	SELECT split(*2017,2018,2019*, ,)
strpos(*string*, *substring*)	BIGINT	返回字符串中第一次出现 *substring* 的位置。位置从 1 开始，如果没有找到子串，则返回 0	SELECT strpos(*prestosql.io*, *.io*);
substr(*string*, *start*, *length*)	VARCHAR	从 *start* 开始去 *length* 长度的子字符串。索引位置从 1 开始。负数索引是从后往前计算的	SELECT substr(*prestosql.io*, 1, 9)
trim(*string*)	VARCHAR	去除头部和尾部空格。与同时使用 *rtrim* 和 *ltrim* 效果一样	SELECT trim(' A ')
upper(*string*)	VARCHAR	将字符串转换为大写	SELECT upper(*lower*)
word_stem(*word*, *lang*)	VARCHAR	返回指定语言下单词的词干	SELECT word_stem(*presto*, *it*)

9.11　字符串和映射

Presto 能够处理映射类型的数据，它有两个有趣的可以返回映射类型的字符串函数：

```
split_to_map(string, entryDelimiter, keyValueDelimiter) → map<varchar, varchar>
```

该函数通过使用 `entryDelimiter` 将字符串参数分割，将字符串拆成包含键值对的字符串，然后使用 `keyValueDelimiter` 将这些字符串拆成键和值，其结果是一个映射。

下面是该函数的一个实例——解析 URL 参数：

```
SELECT split_to_map('userid=1234&reftype=email&device=mobile', '&', '=');
                   _col0
---------------------------------------------
 {device=mobile, userid=1234, reftype=email}
(1 row)
```

当同一个键出现多次时，`split_to_map` 函数会返回错误。

类似的函数 `split_to_multimap` 可以在一个键重复出现时使用。假定前面的例子中有一个重复的 `device` 项：

```
SELECT
split_to_multimap(
  'userid=1234&reftype=email&device=mobile&device=desktop',
  '&',
  '=');
------------------------------------------------------------
 {device=[mobile, desktop], userid=[1234], reftype=[email]}
(1 row)
```

9.12 Unicode

如表 9-8 所示，Presto 提供了一组 **Unicode 函数**。这些函数可以在 UTF-8 有效编码的 Unicode 码点上工作。这些函数区别对待每个码点，即使多个码点表示单个字符时也是如此。

表9-8：与Unicode编码相关的函数

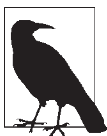
函　数	返回类型	描　述
`chr(n)`	`VARCHAR`	返回 Unicode 码点 *n* 代表的单个字符
`codepoint(string)`	`INTEGER`	返回 *string* 的 Unicode 码点
`normalize(string)`	`VARCHAR`	用 NFC 归一化形式转换字符串
`normalize(string, form)`	`VARCHAR`	用指定的归一化形式转换字符串

`form` 参数必须是表 9-9 中的关键词之一。

表9-9：归一化形式

形　式	描　述
NFD	正则分解
NFC	正则分解，然后正则合成
NFKD	兼容分解
NFKC	兼容分解，然后正则合成

Unicode 标准对这些形式进行了详细的描述。



这个 SQL 标准函数有特殊的语法，需要将 form 作为关键字而不是字符串来指定。

`to_utf8(string) → varbinary`

该函数将字符串编码成 UTF-8 的 varbinary 表示方式。

`from_utf8(binary) → varchar`

该函数从二进制中解码一个 UTF-8 编码的字符串。任何无效的 UTF-8 序列都会被替换为 Unicode 替换字符 U+FFFD。

`from_utf8(binary, replace) → varchar`

该函数从二进制中解码 UTF-8 编码的字符串。任何无效的 UTF-8 序列都会被 *replace* 的内容替换掉。

让我们使用 chr 函数将 Unicode 码点作为字符串返回：

```
SELECT chr(241);
 _col0
-------
 ñ
(1 row)
```

在这个例子中，我们使用函数 codepoint 来返回字符串的 Unicode 码点：

```
SELECT codepoint(u&'\00F1');
 _col0
-------
   241
(1 row)
```

现在我们演示一下西班牙语的 eñe 字符在 Unicode 中用多种方式表示的例子。它既可以用一个码点来表示，也可以由多个码点组成。当直接比较时，它们并不被视为相等的。这时，可以使用归一化函数将它们归一化为共同的形式，以便进行比较并等效处理。

```
SELECT u&'\00F1',
u&'\006E\0303',
u&'\00F1' = u&'\006E\0303',
normalize(u&'\00F1') = normalize(u&'\006E\0303');

 _col0 | _col1 | _col2 | _col3
-------+-------+-------+-------
 ñ     | ñ     | false | true
(1 row)
```

在某些情况下，多个码点可以组成一个码点。例如，罗马数字 IX 可以用 I 和 X 两个码点来写，也可以用单个码点来写。要比较二者的相等性，需要使用归一化函数：

```
SELECT u&'\2168', 'IX', u&'\2168' = 'IX', normalize(u&'\2168', NFKC) = 'IX';
 _col0 | _col1 | _col2 | _col3
-------+-------+-------+-------
 Ⅸ     | IX    | false | true
(1 row)
```

9.13 正则表达式

Presto 通过提供 SQL LIKE 运算符和正则表达式（regex）函数来支持模式匹配。LIKE 返回一个布尔值，语法为*搜索字段* LIKE *模式*。

LIKE 非常适用于只需要基本模式匹配的场合，但在其他场合可能表达力不足。LIKE 模式支持两个符号：_ 表示匹配任何一个字符，% 表示匹配 0 个或多个字符。

假设你想找到从达拉斯地区出发的航班，可以写出下面的查询：

```
SELECT origincityname, count(*)
FROM flights_orc
WHERE origincityname LIKE '%Dallas%'
GROUP BY origincityname;

    origincityname     |  _col1
-----------------------+---------
 Dallas/Fort Worth, TX | 7601863
 Dallas, TX            | 1297795
(2 rows)
```

任何更复杂的模式匹配都需要使用正则表达式函数，这些函数使用 Java 模式语法提供的强大模式匹配能力。表 9-10 中列出的这些函数适用于更复杂的匹配，可以用来替换和提取匹配的字符串，也能够根据匹配的位置分割字符串。表 9-10 列出了这些正则表达式函数。

表9-10：正则表达式函数

函　　数	描　　述
regexp_extract_all(*string*, *pattern*, [*group*]) → array(varchar)	返回在 *string* 中用 *pattern* 匹配到的子字符串数组。该函数的一个变体为接收代表捕获组标号的参数
regexp_extract(*string*, *pattern* [*group*]) → varchar	返回在 *string* 中用 *pattern* 匹配到的子字符串。该函数的一个变体为接收代表捕获组标号的参数
regexp_like(*string*, *pattern*) → boolean	返回一个布尔值，表示 *string* 中是否包含所指定的 *pattern*。和 LIKE 不一样的地方是，LIKE 尝试对整个 *string* 使用 *pattern* 进行匹配
regexp_replace(*string*, *pattern*, [*replacement*]) → varchar	返回一个字符串，其中与 *pattern* 匹配的子字符串被替换为 *replacement*。这个函数的一个变体是省略 *replacement*，这种情况会删除匹配到的字符串。在 *replacement* 字符串中可以使用捕获组
regexp_replace(*string*, *pattern*, *function*) → varchar	这个函数和 regexp_replace(*string*, *pattern*, [*replacement*]) 类似，但它接收的是 lambda 表达式
regexp_split(*string*, *pattern*) → array(varchar)	返回使用 *pattern* 将 *string* 分割后得到的一个数组。pattern 类似于 delimiter，整个函数与 split(*string*, *delimiter*) 类似，但用更具有表达力的正则表达式作为分隔符模式

表 9-11 展示了常用的模式示例。有关正则表达式的 Java 文档中详细地记录了所有受支持的模式。

表9-11：正则表达式的例子

模　式	描　述	示　例
.	任意字符	A
a	单个字符 a	regexp_like(*abc*, *a*) → true
[a-zA-Z]	字符范围	regexp_like(*abc*, *[a-zA-Z]*), regexp_like(*123*, *[a-zA-Z]*)
1	数字 1	regexp_like(*123*, *1*)
\d	任意数字	regexp_like(*123*, *\d*)
^	匹配行首	regexp_like(*abc*, *^ab^*), regexp_like(*abc*, *^bc^*)
$	匹配行尾	regexp_like(*abc*, *bc$*), regexp_like(*abc*, *ab$*)
?	匹配 0 次或 1 次	regexp_like(*abc*, *d?*)
+	匹配 1 次或多次	regexp_like(*abc*, *d+*)
*	匹配 0 次或多次	

在这个例子中，我们想从字符串中提取字符 b。这会返回一个数组，其中每个元素都是一次成功的匹配：

```
SELECT regexp_extract_all('abbbbcccb', 'b');
    _col0
--------------
 [b, b, b, b, b]
(1 row)
```

让我们再次提取字符 b，而这次我们希望提取到 b 的序列。结果集也是一个数组，但只有两个元素，因为其中一个元素包含了 b 的连续序列：

```
SELECT regexp_extract_all('abbbbcccb', 'b+');
 _col0
--------
 [bbbb, b]
(1 row)
```

在下面这个例子中，我们在 replacement 中使用捕获组。我们在寻找 bc 的序列，然后交换其顺序：

```
SELECT regexp_replace('abc', '(b)(c)', '$2$1');
 _col0
-------
 acb
(1 row)
```

9.14 解嵌套复杂数据类型

通过 UNNEST 操作，可以将 8.9.1 节中讨论的复杂集合数据类型展开成关系。对于大数据和嵌套结构数据类型，这是一个非常强大的功能。通过将数据展开到关系中，你可以更容易地在查询中访问嵌套在结构中的值。

例如，你存储了一些访问控制策略，并希望查询这些策略。首先，我们有一个用户，他可以与一个或多个角色关联，而一个角色可以与一个或多个权限集关联。也许这些逻辑是由 schema 定义的，像下面这样：

```
SELECT * FROM permissions;
  user  |                               roles
--------+---------------------------------------------
 matt   | [[WebService_ReadWrite, Storage_ReadWrite],
           [Billing_Read]]
 martin | [[WebService_ReadWrite, Storage_ReadWrite],
           [Billing_ReadWrite, Audit_Read]]
(2 rows)
```

我们可以通过使用 UNNEST 来展开每个角色，然后与用户关联：

```
SELECT user, t.roles
FROM permissions,
UNNEST(permissions.roles) AS t(roles);
  user  |                   roles
--------+------------------------------------------
 martin | [WebService_ReadWrite, Storage_ReadWrite]
 martin | [Billing_ReadWrite, Audit_Read]
 matt   | [WebService_ReadWrite, Storage_ReadWrite]
 matt   | [Billing_Read]
(4 rows)
```

假设我们想过滤数据，找出有 Audit_Read 权限的用户，则可以进一步将结构展开：

```
SELECT user, permission
FROM permissions,
UNNEST(permissions.roles) AS t1(roles),
UNNEST(t1.roles) AS t2(permission);
  user  |      permission
 --------+---------------------
 martin | WebService_ReadWrite
 martin | Storage_ReadWrite
 martin | Billing_ReadWrite
 martin | Audit_Read
 matt   | WebService_ReadWrite
 matt   | Storage_ReadWrite
 matt   | Billing_Read
(7 rows)
```

最后再加上我们的过滤条件：

```
SELECT user, permission
FROM permissions,
UNNEST(permissions.roles) AS t1(roles),
UNNEST(t1.roles) AS t2(permission)
WHERE permission = 'Audit_Read';
  user  | permission
--------+-----------
 martin | Audit_Read
(1 row)
```

9.15 JSON函数

在现代应用程序和系统中，JSON 格式的数据无处不在，应用范围极广。JavaScript Object Notation（JSON）是一种可读性佳且很灵活的数据格式，常用于 Web 应用程序，作为浏览器和服务器之间的数据传输格式。很多需要分析的数据来自于 Web 流量，因此经常使用 JSON 格式输出和保存。专用的文档存储以及许多关系数据库系统现在支持 JSON 数据。

Presto 是一个 SQL-on-Anything 的引擎，它能以 JSON 格式从数据源中检索数据。例如，Kafka、Elasticsearch 和 MongoDB 连接器都返回 JSON 格式的数据，或者可以暴露源 JSON 数据。Presto 也可以使用 HDFS 或云对象存储中的 JSON 文件。Presto 并不强制连接器将 JSON 格式的数据转换为带有列和行的严格的关系数据结构，相反，可以用表 9-12 中的函数对 JSON 数据进行操作，这使得用户可以对原始数据执行他们想要的操作。

表9-12：JSON相关函数

函　数	描　述	示　例
`is_json_scalar(`*json*`)`	如果传入值是 JSON 标量，则返回 `true`	`SELECT is_json_scalar(`*abc*`)`
`json_array_contains(`*json*`, `*value*`)`	如果传入值包含于 JSON 数组中，则返回 `true`	`SELECT json_array_contains([`*1, 2, 3*`], 1)`
`json_array_length(`*json*`)`	返回传入的 JSON 数组的长度（`BIGINT` 类型）	`SELECT json_array_length([`*1, 2, 3*`])`

9.16 日期和时间函数及运算符

在 8.9.2 节中，我们讨论了 Presto 中的时态数据类型。你已经了解了这些类型，知道它们输入和输出的格式和表示方式，以及时区、时间间隔和其他一些细微差别。虽然这些内容涵盖了存储和时态类型的表示，但使用函数和运算符对数据进行操作也很常见且很重要。

Presto 支持时态类型的 `+` 和 `-` 运算符。这些运算符可以用来在日期时间类型上加减一个时间间隔，或者在时间间隔类型上加减另一个时间间隔。但是，将两个时间戳加在一起没有任何意义。此外，`YEAR TO MONTH` 和 `DAY TO SECOND` 的时间间隔类型不能合并。

你可以在时间 12:00 的基础上加 1 小时，得到 13:00：

```
SELECT TIME '12:00' + INTERVAL '1' HOUR;
    _col0
--------------
 13:00:00.000
(1 row)
```

接下来，你可以把 1 年和 15 个月加在一起，结果为 2 年零 3 个月：

```
SELECT INTERVAL '1' YEAR + INTERVAL '15' MONTH;
 _col0
-------
 2-3
(1 row)
```

另一个好用的运算符 AT TIME ZONE 允许你计算不同时区的时间：

```
SELECT TIME '02:56:15 UTC' AT TIME ZONE '-08:00'
        _col0
---------------------
 18:56:15.000 -08:00
(1 row)
```

只要使用正确的格式，就可以将字符串解析成时间戳类型的值：

```
SELECT TIMESTAMP '1983-10-19 07:30:05.123';
          _col0
-------------------------
 1983-10-19 07:30:05.123
```

在许多情况下，使用 ISO 8601 格式和表 9-13 中的函数之一来解析字符串中的日期或时间戳更方便。

表9-13：ISO 8601解析函数

函　　数	返回类型	描　　述
from_iso8601_timestamp(*string*)	TIME WITH TIME ZONE	解析一个 ISO 8601 格式的字符串并返回一个 TIME WITH TIME ZONE 类型的结果
from_iso8601_date(*string*)	DATE	解析一个 ISO 8601 格式的字符串返回一个 DATE 类型的结果

ISO 8601 是一个关于时间字符串格式的很好的标准。在使用前面列出的函数时，传入的字符串必须使用以下格式之一：

- YYYY
- YYYY-MM
- YYYY-MM-DD
- HH

- HH:MM
- HH:MM:SS
- HH:MM:SS.SSS

此外，还可以使用 T 分隔符将日期和时间结合起来，下面是一些例子。

我们先将 ISO 8601 日期和时间解析成 SQL 时间戳：

```
SELECT from_iso8601_timestamp('2019-03-17T21:05:19Z');
            _col0
-----------------------------
 2019-03-17 21:05:19.000 UTC
(1 row)
```

接下来指定一个默认的 UTC 以外的时区：

```
SELECT from_iso8601_timestamp('2019-03-17T21:05:19-05:00');
             _col0
--------------------------------
 2019-03-17 21:05:19.000 -05:00
(1 row)
```

该标准还允许你指定一年中的某周。在下面这个例子中，2019 年的第 10 周相当于 2019 年 3 月 4 日：

```
SELECT from_iso8601_timestamp('2019-W10');
            _col0
-----------------------------
 2019-03-04 00:00:00.000 UTC
(1 row)
```

在下面这个例子中，我们不指定时间，而是将 ISO 8601 字符串解析为 SQL DATE 类型：

```
SELECT from_iso8601_date('2019-03-17');
   _col0
------------
 2019-03-17
(1 row)
```

Presto 提供了一组丰富的与日期和时间相关的函数。这些函数对于涉及时间的应用程序至关重要，在这些应用程序中，你可能经常要转换、比较或提取时间元素。表 9-14 展示了一些可用的函数。请查看 1.4.2 节了解更多有用的函数和其他技巧。

表9-14：一些时态函数和值

函　数	返回类型	描　述
current_timezone()	VARCHAR	返回当前时区
current_date	DATE	返回当前日期
current_time	TIME WITH TIME ZONE	返回当前日期和时区

（续）

函　　数	返回类型	描　　述
`current_timestamp or now()`	`TIMESTAMP WITH TIME ZONE`	返回当前时间、日期和时区
`localtime`	`TIME`	基于本地时区返回当前时间
`localtimestamp`	`TIMESTAMP`	基于本地时区返回当前的日期和时间
`from_unixtime(unixtime)`	`TIMESTAMP`	将一个 UNIX 时间转换为日期和时间
`to_unixtime(timestamp)`	`DOUBLE`	将日期和时间转换为 UNIX 时间值
`to_milliseconds(interval)`	`BIGINT`	将时间间隔转换为毫秒

9.17　直方图

Presto 提供了 `width_bucket` 函数用来创建等宽**直方图**。

```
width_bucket(x, bound1, bound2, n) -> bigint
```

表达式 *x* 表示用来创建直方图的数值。等宽直方图包含 *n* 个桶，并使用 *bound1* 和 *bound2* 作为值的界限。该函数返回表达式 *x* 的每个值对应的桶编号。

让我们以航班数据集为例，计算出从 2010 年到 2020 年 10 年内的直方图：

```
SELECT count(*) count, year, width_bucket(year, 2010, 2020, 4) bucket
FROM flights_orc
WHERE year >= 2010
GROUP BY year;

  count  | year | bucket
---------+------+--------
 7129270 | 2010 |      0
 7140596 | 2011 |      1
 7141922 | 2012 |      1
 7455458 | 2013 |      1
 7009726 | 2014 |      2
 6450285 | 2015 |      2
 6450117 | 2016 |      3
 6085281 | 2017 |      3
 6096762 | 2018 |      3
 6369482 | 2019 |      4
(10 rows)
```

注意，除了预期的桶 1、桶 2、桶 3 和桶 4 外，我们还有桶 0 和桶 5[1]。这些桶用来放最小值和最大值范围以外的值，在本例中，也就是 2010 年到 2019 年以外的年份。

注 1：本例中没有出现 2020 年的数据，因此没有显示桶 5。——译者注

9.18 聚合函数

在 SQL 中，**聚合函数**操作并计算出一个值或一组值。与标量函数对每个输入值产生单个值不同，聚合函数对一组输入值产生单个值。Presto 支持大多数其他数据库系统中常见的通用聚合函数，如表 9-15 所示。

聚合函数可以在参数后加入一个可选的 `ORDER BY` 子句。在语义上，这意味着在执行聚合之前对输入集进行排序。对于大多数聚合函数来说，顺序并不重要。

表9-15：聚合函数

函　数	返回类型	描　述
`count(*)`	`BIGINT`	返回传入数值的个数
`count(x)`	`BIGINT`	返回传入非 `NULL` 值的个数
`sum(x)`	同输入	返回传入值的总和
`min(x)`	同输入	返回传入值的最小值
`max(x)`	同输入	返回传入值的最大值
`avg(x)`	`DOUBLE`	返回传入值的平均值

9.18.1 映射聚合函数

Presto 支持几个有用的映射相关函数，见表 9-16。对于其中一些函数，根据所需的结果，需要使用可选的 `ORDER BY` 子句。我们用鸢尾花数据集（参见 1.4.7 节）的例子来演示这种使用方法。

表9-16：映射聚合函数

函　数	返回类型	描　述
`histogram(x)`	`map(K, bigint)`	这个函数从传入值 *x* 中创建一个直方图，返回一个映射，其中键是 *x*，值是 *x* 出现的次数
`map_agg(key, value)`	`map(K, V)`	从一组键值中创建映射，重复键对应的值是随机选取的，可以使用 `multimap_agg` 来生成单键对多值的映射
`map_union(x(K, V))`	`map(K, V)`	这个函数将多个映射合并到一个映射中。需要注意的是，如果在多个映射中找到相同的键，则不能保证哪个值会被选择。该函数不会合并这两个值
`multimap_agg(key, value)`	`map(K, array(V))`	此函数类似于 `map_agg`，它也是从键值中创建的映射[2]

让我们创建一个 `petal_length_cm` 的直方图。因为数据是精确的，所以可以使用 `floor` 函数为直方图创建更宽的桶：

注 2：不同之处在于该函数对于同一键会保留出现的多个值。——译者注

```
SELECT histogram(floor(petal_length_cm))
FROM memory.default.iris;
                  _col0
------------------------------------------
 {1.0=50, 4.0=43, 5.0=35, 3.0=11, 6.0=11}
(1 row)
```

你可能认识到，直方图的输出与你在做 GROUP BY 和 COUNT 时看到的结果类似。我们可以将结果作为 map_agg 函数的输入，用同样的结果创建直方图：

```
SELECT floor(petal_length_cm) k, count(*) v
FROM memory.default.iris
GROUP BY 1
ORDER BY 2 DESC;
  k  | v
-----+----
 1.0 | 50
 4.0 | 43
 5.0 | 35
 3.0 | 11
 6.0 | 11
(5 rows)

SELECT map_agg(k, v) FROM (
  SELECT floor(petal_length_cm) k,
    count(*) v
  FROM iris
  GROUP BY 1
);
                  _col0
------------------------------------------
 {4.0=43, 1.0=50, 5.0=35, 3.0=11, 6.0=11}
(1 row)

SELECT multimap_agg(species, petal_length_cm)
FROM memory.default.iris;
----------------------------------------------------
 {versicolor=[4.7, 4.5, 4.9..], ,
 virginica=[6.0, 5.1, 5.9, ..],
 setosa=[1.4, 1.4, 1...] ..
(1 row)
```

map_union 函数对组合映射非常有用。假设有多个映射，本例将使用直方图函数来创建它们：

```
SELECT histogram(floor(petal_length_cm)) x
FROM memory.default.iris
GROUP BY species;
             x
-------------------------
 {4.0=6, 5.0=33, 6.0=11}
 {4.0=37, 5.0=2, 3.0=11}
 {1.0=50}
(3 rows)
```

我们可以用 map_union 来组合它们。然而，注意键 4.0 和键 5.0 在不同的映射中都存在。在这些情况下，Presto 会任意选择一组值。它并没有执行任何类型的合并。虽然在本例中将他们加起来是正确的，但这个处理方式并不总是管用的。例如，这些值可能是字符串，这样就不太清楚如何合并它们了。

```
SELECT map_union(m)
FROM (
  SELECT histogram(floor(petal_length_cm)) m
  FROM memory.default.iris
  GROUP BY species
  );
                  _col0
-----------------------------------------
 {4.0=6, 1.0=50, 5.0=33, 6.0=11, 3.0=11}
(1 row)
```

9.18.2 近似聚合函数

当处理大量的数据和聚合时，数据处理可能是资源密集型的，需要更多的硬件来扩展 Presto 以提供交互性。有时，扩容的成本非常昂贵。

为了帮助应对这种情况，Presto 提供了一组聚合函数，返回近似值而不是精确结果。这些**近似聚合函数**使用更少的内存和计算能力，但代价是不能提供准确的结果。

在处理大数据时，很多情况下这是可以接受的，因为数据本身往往不是完全准确的。可能会缺失一天的数据，但考虑一年以上的聚合时，缺失的数据对于某些类型的分析来说并不重要。

记住，Presto 不是为 OLTP 查询而设计的，它不适合为公司的分类账生成绝对准确的报表。然而，对于 OLAP 使用场景，只需了解趋势而无须完全准确的结果分析，使用 Presto 是可以接受并满足要求的。

Presto 提供了两个主要的近似函数：approx_distinct 和 approx_percentile。

Presto 使用 HyperLogLog 算法来实现 approx_distinct 函数。在数据分析中，统计不同行的数量是一个非常常见的查询。例如，你可能想统计日志的不同用户 ID 或 IP 地址的数量，以了解某天、某月或某年有多少用户访问了你的网站。由于用户可能多次访问你的网站，因此，简单地统计日志中的条目数是行不通的。你需要找到不同用户数，这就要求你将每个用户的多次访问记录仅计为一次。为此，你需要在内存中保存一个结构，这样就可以不用重复计算了。对于海量数据，这种方法就变得不切实际并且很慢，而 HyperLogLog 提供了另一种方法，本书不会讨论具体算法。

为了实现近似的不同值[3]统计，Presto 提供了 HyperLogLog 数据类型（用户也可以使用它），这意味着它可以存入表中。这非常有价值，因为你可以存储计算结果，以便以后再合并。假设你的数据是按天划分的，每天你可以为当天的用户创建一个 HyperLogLog 结构并存储。然后，当你想计算近似基数时，可以将 HyperLogLog 合并到一起得出结果。

9.19 窗函数

Presto 支持使用 SQL 中的标准**窗函数**（window function），它允许你定义一组记录作为函数的输入。

例如，我们来看看鸢尾花的萼片长度（参见 1.4.7 节）。

如果没有窗函数，可以按如下方式计算所有品种的萼片平均长度：

```
SELECT avg(sepal_length_cm)
FROM memory.default.iris;
5.8433332
```

另外，你也可以计算出特定品种的平均长度：

```
SELECT avg(sepal_length_cm)
FROM memory.default.iris
WHERE species = 'setosa';
5.006
```

但是，如果你想列出所有测量值，并将每个测量值与整体平均数比较呢？ `OVER()` 窗函数可以让你做到这一点：

```
SELECT species, sepal_length_cm,
  avg(sepal_length_cm) OVER() AS avgsepal
FROM memory.default.iris;
  species   | sepal_length_cm | avgsepal
------------+-----------------+-----------
 setosa     |             5.1 | 5.8433332
 setosa     |             4.9 | 5.8433332
 ...
 versicolor |             7.0 | 5.8433332
 versicolor |             6.4 | 5.8433332
 versicolor |             6.9 | 5.8433332
...
```

该窗函数表示在同一个表中计算所有值的平均值。你也可以用 `PARTITION BY` 语句创建多个窗：

注 3：不同值的数量常叫作基数。——译者注

```
SELECT species, sepal_length_cm,
  avg(sepal_length_cm) OVER(PARTITION BY species) AS avgsepal
FROM memory.default.iris;
  species   | sepal_length_cm | avgsepal
------------+-----------------+----------
 setosa     |             5.1 |    5.006
 setosa     |             4.9 |    5.006
 setosa     |             4.7 |    5.006
...
 virginica  |             6.3 |    6.588
 virginica  |             5.8 |    6.588
 virginica  |             7.1 |    6.588
...
```

平均长度因品种而异。借助 DISTINCT，通过省略单个长度，你可以得到每个品种的平均长度列表：

```
SELECT DISTINCT species,
  avg(sepal_length_cm) OVER(PARTITION BY species) AS avgsepal
FROM memory.default.iris;
  species   | avgsepal
------------+----------
 setosa     |    5.006
 virginica  |    6.588
 versicolor |    5.936
(3 rows)
```

Presto 中的窗函数支持所有的聚合函数以及许多窗特有的函数：

```
SELECT DISTINCT species,
  min(sepal_length_cm) OVER(PARTITION BY species) AS minsepal,
  avg(sepal_length_cm) OVER(PARTITION BY species) AS avgsepal,
  max(sepal_length_cm) OVER(PARTITION BY species) AS maxsepal
FROM memory.default.iris;
  species   | minsepal | avgsepal | maxsepal
------------+----------+----------+----------
 virginica  |      4.9 |    6.588 |      7.9
 setosa     |      4.3 |    5.006 |      5.8
 versicolor |      4.9 |    5.936 |      7.0
(3 rows)
```

更多细节可以查看 Presto 文档（参见 1.4.2 节）。

9.20 lambda表达式

在 SQL 语句中使用 lambda **表达式**是处理数组元素的一个高级概念。如果你有编程背景，可能对它们的语法很熟悉，或者知道这些表达式的其他名称，如 lambda 函数、匿名函数或闭包。

许多数组函数，如 zip_with，支持使用 lambda 表达式。该表达式定义了一个从输入值到

输出值的转换，用 -> 隔开：

```
x -> x + 1
(x, y) -> x + y
x -> IF(x > 0, x, -x)
x -> x+1 (x, y) -> x + y x -> IF(x > 0, x, -x)
```

其他常用到 lambda 表达式的函数有 `transform`、`filter`、`reduce`、`array_sort`、`none_match`、`any_match` 和 `all_match` 等。

让我们看下面这个例子：

```
SELECT zip_with(ARRAY[1, 2, 6, 2, 5],
                ARRAY[3, 4, 2, 5, 7],
                (x, y) -> x + y);

[4, 6, 8, 7, 12]
```

如你所见，lambda 表达式很简单但功能强大。它将两个数组中相同位置的元素相加，用结果创建一个新的数组。简单来说，对两个数组同时进行迭代，每次迭代时都会调用该函数，而无须编写代码来循环遍历数组数据结构。

9.21 地理空间函数

Presto 所支持的 SQL 超出了标准 SQL 的范围，它包括了地理空间分析领域的一系列重要函数。与其他支持的 SQL 特性一样，Presto 在这方面与其他工具的相关标准和使用方法尽量保持一致。

在**地理空间函数**方面，Presto 使用 `ST_` 前缀支持 SQL/MM 规范和开放地理空间信息联盟（OGC）的 OpenGIS 规范。

因为对地理空间的 SQL 支持内容很多，所以本节只对此进行粗略的介绍。

Presto 支持众多的构造器从数据源处来创建地理空间对象：

- `ST_GeometryFromText(varchar) -> Geometry`
- `ST_GeomFromBinary(varbinary) -> Geometry`
- `ST_LineFromText(varchar) -> LineString`
- `ST_LineString(array(Point)) -> LineString`
- `ST_Point(double, double) -> Point`
- `ST_Polygon(varchar) -> Polygon`

然后，可以在诸多函数中使用这些对象来进行位置比较等处理：

- `ST_Contains(Geometry, Geometry) -> boolean`
- `ST_Touches(Geometry, Geometry) -> boolean`
- `ST_Within(Geometry, Geometry) -> boolean`
- `ST_Length(Geometry) -> double`
- `ST_Distance(Geometry, Geometry) -> double`
- `ST_Area(SphericalGeography) -> double`

Presto 文档（参见 1.4.2 节）详细介绍了 Presto 中对地理空间的支持。如果你在使用 Presto 时需要处理地理空间数据，我们强烈建议你看一下该文档。

9.22 Prepared Statement

Prepared Statement 是一种非常有用的方法，它可以使用不同的输入参数值运行相同的 SQL 语句。它允许重复使用 SQL 语句，这简化了用户反复调用的麻烦，也让代码更干净、更易于维护。Prepared Statement 是保存在 Presto 用户会话中的查询。

Prepared Statement 的使用和创建分为两个步骤。`PREPARE` 语句用于创建语句，从而使其在会话中可重复使用：

```
PREPARE example
FROM SELECT count(*) FROM hive.ontime.flights_orc;
```

`EXECUTE` 命令可用于运行一次或多次查询：

```
EXECUTE example;
   _col0
-----------
 166628027
(1 row)
```

Prepared Statement 可以支持在执行时传入参数值：

```
PREPARE delay_query FROM
SELECT dayofweek,
       avg(depdelayminutes) AS delay
FROM flights_orc
WHERE month = ?
AND origincityname LIKE ?
GROUP BY dayofweek
ORDER BY dayofweek;
```

执行带参数的查询时，需要在 `USING` 关键字之后以正确的顺序传递参数。

```
EXECUTE delay_query USING 2, '%Boston%';
 dayofweek |       delay
-----------+-------------------
```

```
        1 | 10.613156692553677
        2 |   9.97405624214174
        3 |  9.548045977011494
        4 | 11.822725778003647
        5 | 15.875475113122173
        6 | 11.184173669467787
        7 | 10.788121285791464
(7 rows)
```

> **PREPARE 语句在 RDBMS 中与其在 Presto 中的区别**
>
> 在其他关系数据库系统中使用 PREPARE 的目的不仅仅是方便执行传入不同参数值的类似查询。很多系统在执行 PREPARE 语句时，实际上可能会对 SQL 进行解析和计划。然后在执行 EXECUTE 命令时，将这些值传递给系统，并绑定到执行计划算子中。对于事务性系统来说，这往往是重要的优化，因为即使传入不同的值，多次执行查询时只需一次解析和生成执行计划即可。这就是 Prepared Statement 最初的目的。
>
> 另一个常见的例子是 INSERT 查询，在这里你想尽快插入大量的新值。PREPARE 消除了每次插入时生成执行计划的开销。
>
> 目前，Presto 没有实现这种优化，每次 EXECUTE 时都要为查询和解析生成计划。考虑到 Presto 的主要使用场景，上述的绑定优化便并不是那么值得关注了。Prepared Statement 最初是为 JDBC 驱动和 ODBC 驱动而引入 Presto 中的，因为很多工具可能会依赖这个功能。

DESCRIBE 命令可以用 DESCRIBE INPUT 命令和 DESCRIBE OUTPUT 命令来理解 Prepared Statement。这些命令被 JDBC 和 ODBC 驱动内部用于获得元数据信息和错误处理。

```
DESCRIBE INPUT delay_query;
 Position |  Type
----------+---------
        0 | integer
        1 | varchar
(2 rows)

DESCRIBE OUTPUT delay_query;
 Column Name | Catalog | Schema |    Table     |  Type   | Type Size | Aliased
-------------+---------+--------+--------------+---------+-----------+--------
 dayofweek   | hive    | ontime | flights_orc  | integer |         4 | false
 delay       |         |        |              | double  |         8 | true
(2 rows)
```

当你退出 Presto 会话时，Prepared Statement 会被自动释放。你可以用 DEALLOCATE PREPARE 语句手动释放 Prepared Statement。

```
DEALLOCATE PREPARE delay_query;
DEALLOCATE
```

9.23 小结

恭喜你学完了本章！这一章比较深入地讲解了 Presto 中的 SQL 支持和处理。这是 Presto 的一个核心功能，因此非常重要。不过你并没有学到所有的细节，请务必参考 1.4.2 节中提到的官方文档，以获得全面最新的信息，包括所有函数和运算符的完整列表以及更多的细节。

希望深入学习 Presto 的 SQL 支持能让你真正在实际生产环境中运行 Presto 时为用户创造价值。在本书的第三部分，你将了解到更多关于安全、监控等方面的内容，还可以找到一些使用 Presto 的应用实例，以及其他组织和公司的实际使用情况。

有了第 8 章和本章中关于 Presto 的 SQL 支持的知识，你现在可以回到第 4 章，学习更多关于查询计划和优化的知识；你也可以进入本书的第三部分，了解更多关于 Presto 在生产中环境的使用、集成 Presto 的方法以及其他使用场景。

第三部分

Presto的实际应用

到目前为止，你已简要了解了 Presto，学习了如何在生产环境中安装 Presto，掌握了如何用不同的连接器连接数据源，并看到了它的 SQL 支持是多么强大。

在第三部分中，你将学习在生产环境中使用 Presto 的其他方面，如安全和监控；你将探索与 Presto 一起工作的应用程序，从而为用户提供巨大的价值。

最后，你会了解到其他组织使用 Presto 的情况。

第10章 安全

第 5 章中讨论的大规模部署是实现在生产环境中使用 Presto 的重要一步。在本章中，你将了解更多关于 Presto 本身以及底层数据安全问题的信息。

在典型的 Presto 集群部署和使用中，可以考虑保障几个方面的安全：

- 用户客户端与 Presto 协调器之间的数据传输；
- Presto 集群内协调器和工作节点之间的数据传输；
- Presto 集群和按 catalog 配置的每个数据源之间的数据传输；
- 访问每个数据源内的具体数据。

在图 10-1 中，你可以看到如何保障 Presto 不同网络连接的安全性：与客户端的连接（比如，Presto CLI 或使用 JDBC 驱动的应用程序）、集群内的流量，以及与所有不同数据源的连接都需要安全保障。

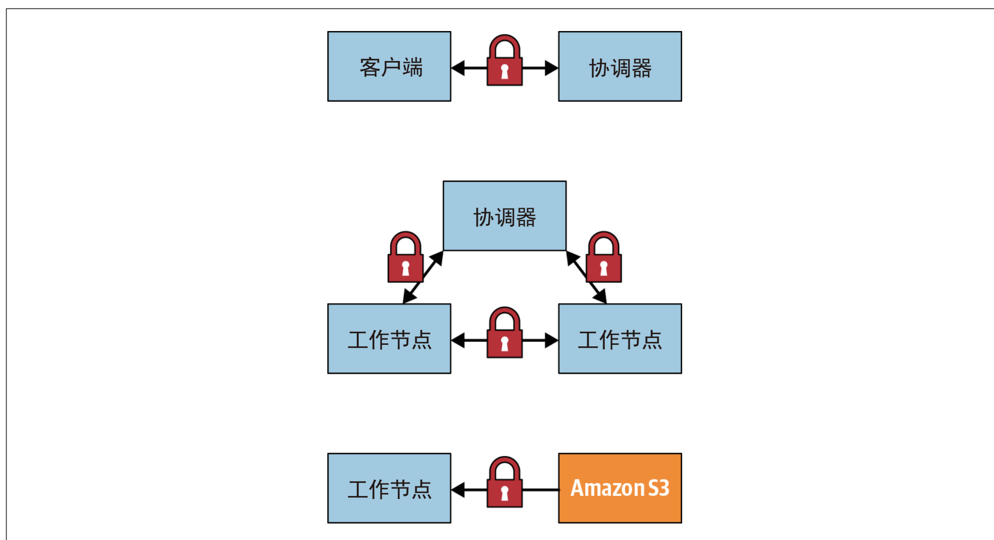


图 10-1：需要安全保障的 Presto 网络连接

下面各节会更详细地探讨这些需求，先从 Presto 的用户认证开始。

10.1　认证

认证是向系统证明身份的过程，它对于任何安全系统都必不可少。计算机系统支持多种认证方法，包括 Kerberos、轻量级目录访问协议（LDAP）密码认证以及证书认证。是否支持特定的认证方法取决于每个系统。在 Presto 中，客户端通常通过以下方法之一来进行 Presto 的认证。

- 通过 LDAP 输入密码，参见下述内容。
- 证书，参见 10.5 节。
- Kerberos，参见 10.6 节。

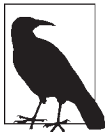
默认情况下，Presto 中没有配置认证。任何可以访问协调器的人都可以连接并执行查询或其他操作。在本章中，你将学习有关 LDAP 和证书认证机制的详细信息，因为它们是最常用的。

然而，认证只是其中的一个环节。一旦安全主体通过了认证，他们就会分配到一定的权限，这些权限决定了用户能够做什么。你能做的事情称为**授权**，由 `SystemAccessControl` 和 `ConnectorAccessControl` 管辖。10.2 节将详细介绍相关信息。下面的假设一旦认证后，用户就可以执行任何操作。实际上，以系统权限访问系统时，默认就可以执行任何操作，如查询系统 catalog。

密码认证和LDAP认证

密码认证是一种你可能每天都在使用的认证方法。通过给系统提供用户名和密码，你提供了只有你知道的信息来证明身份。Presto 通过密码验证模块来支持这种基本的验证方式。**密码验证模块**接收来自客户端的用户名和密码凭证，对其进行验证，并创建一个安全主体。密码验证模块旨在支持作为插件部署的自定义密码验证模块。

目前，Presto 的密码认证使用的是基于 LDAP 服务的 LDAP 验证器。



Presto 支持密码文件认证，这是一种简单但不太常用的认证机制。

LDAP 是一种业界标准的应用协议，用于访问和管理目录服务器中的信息。目录服务器中的数据模型是一个存储身份信息的分层数据结构。关于什么是 LDAP 及其工作原理，这里不再赘述，你可以在其他书中或网上找到更多信息。

当使用 LDAP 验证模块时，用户将用户名和密码传给 Presto 协调器，这可以通过 CLI、JDBC 驱动或任何其他支持传递用户名和密码的客户端来完成。然后，协调器通过外部 LDAP 服务验证这些凭证，并从用户名中创建安全主体。你可以在图 10-2 中看到这个流程。要在 Presto 中启用 LDAP 认证，需要在 Presto 协调器的 config.properties 文件中添加：

```
http-server.authentication.type=PASSWORD
```

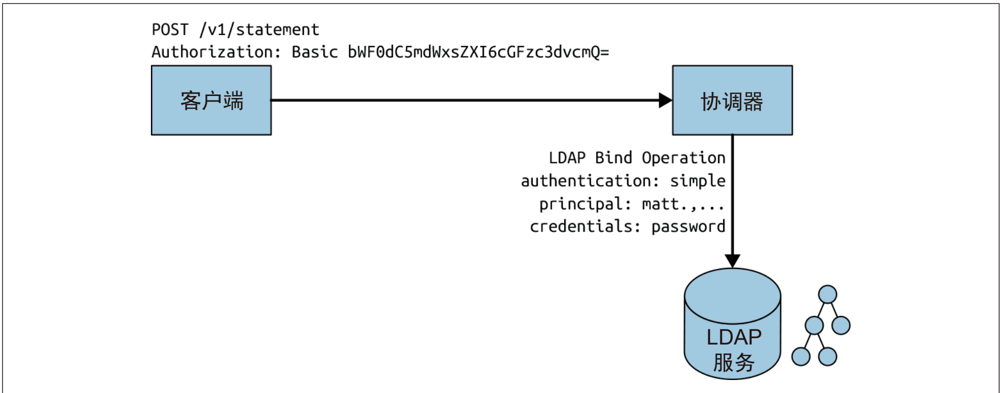


图 10-2：使用外部 LDAP 服务对 Presto 进行 LDAP 认证

通过设置认证类型为 `PASSWORD`，我们告诉 Presto 协调器使用密码验证模块进行认证。

此外，你需要在 etc 目录下另外添加一个文件 password-authenticator.properties 来配置 LDAP 服务：

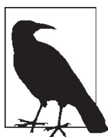
```
password-authenticator.name=ldap
ldap.url=ldaps://ldap-server:636
ldap.user-bind-pattern=${USER}@example.com
password-authenticator.name=ldap
ldap.url=ldaps:/ldap-server:636
ldap.user-bind-pattern=${USER}@example.com
```

password-authenticator.name 属性指定了使用 LDAP 插件进行密码验证，随后的几行则配置了 LDAP 服务器的 URL 和定位用户记录的模式。上述绑定模式是与 Active Directory 配合使用时的示例用法，下面是使用其他 LDAP 服务器或配置定义用户 ID（UID）的另一个示例：

```
ldap.user-bind-pattern=uid=${USER},OU=people,DC=example,DC=com
```

在 LDAP 中，添加、删除、修改、搜索和绑定等操作类型与目录交互。**绑定**是验证客户端到目录服务器的操作，也是 Presto 用于支持 LDAP 认证的操作。为了绑定，你需要提供身份标识和身份证明，如密码。LDAP 允许不同类型的认证，但用户身份标识（也称为**可分辨名称**）和密码是 Presto 支持 LDAP 认证的主要方法。

Presto 最近增加了一种辅助方法，就是用 LDAP 服务用户进行认证和授权，可以查看 Presto 文档中的相关配置提示。



Presto 需要使用**安全的 LDAP**，也就是 **LDAPS**。因此，你需要确保 LDAP 服务器上启用了 TLS，还需要确保 ldap-url 属性的 URL 使用 ldaps:// 而非 ldap://。因为这个通信是通过 TLS 进行的，所以你需要将 LDAP 服务器的 TLS 证书导入 Presto 协调器使用的 truststore 中。或者如果 LDAP 服务器使用的是由认证中心（certificate authority，CA）签署的证书，则你需要确保 CA 链是在 truststore 中的。

为了安全地使用 LDAP 验证模块，你还需要配置对 Presto 协调器的 HTTPS 访问，这可以确保从客户端发送的密码不会以明文的形式在不安全的网络上传输。10.3 节将讨论此设置。

LDAP 配置好后，你可以使用 Presto CLI 进行测试：

```
presto --user matt --password
```

指定 --password 时，CLI 会提示你输入密码。配置 LDAP 密码验证模块时，你已经设置了用户绑定模式。

当用户名和密码从客户端（也就是我们例子中的 CLI）传给 Presto 协调器时，Presto 会在绑定模式中填入用户名作为安全主体，并将密码作为安全凭证，将它们作为 LDAP 绑定请求的一部分发送出去。在我们的例子中，用于匹配目录结构中的可分辨名称的安全主体是 uid=matt,OU=people,DC=example,DC=com。LDAP 目录中的每个条目可能由多个属性组成。

其中一个属性是 `userPassword`，绑定操作使用该属性来匹配发送的密码。

Presto 可以根据组成员身份进一步限制访问。你可以在 10.2 节中了解更多关于组的重要性。通过使用组，你可以给一个组分配权限，因此该组中的用户可以继承该组的所有权限，而无须单独管理权限。比如，你想只允许 `engineering` 组中的用户使用 Presto。在我们的例子中，我们希望用户 matt 和 maria 而非用户 jane 能够访问 Presto。

为了进一步根据组成员资格限制用户，Presto 允许你在 password-authenticator.properties 文件中指定额外的属性。

```
ldap.user-base-dn=OU=people,DC=example,DC=com

ldap.group-auth-pattern=(&(objectClass=inetOrgPerson)(uid=${USER})(memberof=CN=
developers,OU=groups,DC=example,DC=com))
```

在我们的例子中，前面的过滤器在基础可分辨名称中限制了用户，只允许属于 LDAP 中的 `developers` 组的用户访问。如果用户 jane 试图验证，绑定操作会成功，因为 jane 是一个有效的用户，但这个用户终将被过滤掉，因为她不属于 `developers` 组。

10.2 授权

在 10.1 节中，你了解了认证，或者说向 Presto 证明你是谁。但是，在一个有很多用户和敏感数据的环境中，你并不希望通过认证的任何用户都能访问所有数据。

为了限制访问，你需要配置用户可以做什么——**授权**。让我们先来研究一下 Presto 中的 SQL 模型，看看有哪些访问控制。然后，你将学习如何在系统和连接器层面控制 Presto 的访问权限。

10.2.1 系统访问控制

系统访问控制在 Presto 全局级别上执行授权，并允许你配置 catalog 的访问权限和安全主体所使用的规则。

catalog 内更细粒度的权限和限制必须通过连接器访问控制进行配置，参见 10.2.2 节。

如 10.1 节所述，安全主体是用于通过 Presto 的验证的实体。安全主体可以是单个用户或服务账户。Presto 还将用于认证的安全主体与运行查询的用户分开。例如，多个用户可以共享一个安全主体进行认证，但以自己的身份运行查询。默认情况下，Presto 允许任何可以认证的安全主体以其他人的身份运行查询：

```
$ presto --krb5-principal alice@example.com --user bob
```

这通常不是你在真实环境中想要的，因为它可能会使一个用户（bob）的访问权限超出其授

权范围。改变这种默认行为需要额外的配置，而 Presto 支持一组内置的系统访问控制配置。

默认情况下，Presto 允许经过认证的所有用户做任何事情。这是最不安全的配置，不建议在生产环境中部署时使用。虽然是默认配置，但你仍可以在 etc 目录下创建一个 access-control.properties 文件来显式地设置它。

```
access-control.name=allow-all
```

只读授权稍微安全一些，因为它只允许任何读取数据或元数据的操作。这包括 SELECT 查询，但不包括 CREATE、INSERT 和 DELETE 查询。

```
access-control.name=read-only
```



使用这种方法将访问控制设置为只读，是一种非常快速、简单且有效的方法，可以降低 Presto 使用底层数据的风险。同时，只读访问完全适用于分析使用，你可以轻松地创建一个专用的 Presto 集群，允许公司中的任何人访问大量的数据来进行数据分析、问题数据排查，或者仅仅是探索和学习更多 Presto 的知识。

要配置系统访问控制，除了简单的 allow-all 和 read-only 之外，还可以使用基于文件的方法。这可以让你指定用户对 catalog 的访问控制规则，以及安全主体可以识别为哪些用户。这些规则是在你维护的文件中指定的。

当使用基于文件的系统访问控制时，所有对 catalog 的访问都会被拒绝，除非有匹配的规则明确赋予用户权限。你可以在 etc 配置文件中的 access-control.properties 文件中启用基于文件的访问控制：

```
access-control.name=file
security.config-file=etc/rules.json
```

security.config-file 属性指定了规则文件的位置。它必须是一个 JSON 文件，使用下面代码中详述的格式。建议将其与其他所有配置文件存放在同一目录下。

```
{
  "catalogs": [
    {
      "user": "admin",
      "catalog": "system",
      "allow": true
    },
    {
      "catalog": "hive",
      "allow": true
    },
    {
      "user": "alice",
```

```
        "catalog": "postgresql",
        "allow": true
      }
      {
        "catalog": "system",
        "allow": false
      }
    ]
  }
```

Presto 会按顺序检查规则，并使用第一个匹配的规则。在本例中，管理员用户可以访问 `system` catalog，而最后一条规则使其他所有用户无法访问它。前面提到，除非有匹配的规则，否则所有的 catalog 访问都会被默认拒绝，但例外的情况是，所有用户默认都有访问 `system` catalog 的权限。

该示例文件还授予了所有用户对 `hive` catalog 的访问权限，但只有用户 alice 被授予了对 `postgresql` catalog 的访问权限。



系统访问控制对于限制访问是非常有用的。但是，它们只能在 catalog 级配置访问权限，不能用于配置更细粒度的访问权限。

如前所述，经过认证的安全主体可以默认以任何用户的身份运行查询。一般不建议这样做，因为这允许用户以其他人的身份访问数据。如果连接器实现了一个连接器访问控制，这意味着用户可以用通过认证的安全主体冒充另一个用户来访问他本不能访问的数据。因此，必须在安全主体和运行查询的用户之间执行适当的匹配。

假设我们想把用户名设置为 LDAP 安全主体的名称：

```
{
  "catalogs": [
    {
      "allow": "all"
    }
  ],
  "principals": [
    {
      "principal": "(.*)",
      "principal_to_user": "$1",
      "allow": "all"
    }
  ]
}
```

可以进一步扩展到强制用户使用其 Kerberos 安全主体名称。此外，我们还可以将用户名匹配到可能共享的组安全主体：

```
"principals": [
  {
    "principal": "([^/]+)/?.*@example.com",
    "principal_to_user": "$1",
    "allow": "all"
  },
  {
    "principal": "group@example.com",
    "user": "alice|bob",
    "allow": "all"
  }
]
```

因此，你必须覆盖相关规则以获得不同的行为。在这种情况下，用户 bob 和 alice 可以使用安全主体 group@example.com，也可以使用自己的安全主体 bob@example.com 和 alice@example.com。

10.2.2　连接器访问控制

回顾一下，Presto 为了查询数据而暴露的那些对象集合。catalog 是连接器的配置实例。一个 catalog 可能由一组命名空间组成，称为 schema，而 schema 包含了一组使用某些数据类型和数据行的表。通过连接器访问控制，Presto 允许你在 catalog 中配置细粒度的权限。

Presto 支持标准 SQL 中的 GRANT 语句来授予用户或角色在表和视图上的权限，也支持将用户成员资格授予角色。如今，Presto 支持 SQL 标准所定义的权限的一个子集。在 Presto 中，你可以向一个表或视图授予以下权限。

SELECT
: 相当于读取权限。

INSERT
: 相当于创建或写入新行的数据权限。

DELETE
: 相当于删除行的数据权限。



撰写本书时，只有 Hive 连接器支持角色和授权。这一功能基于连接器的实现，所以每个连接器都需要实现 ConnectorAccessControl 来支持这个标准 SQL 功能。

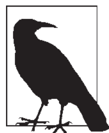
我们来看一个例子：

```
GRANT SELECT on hive.ontime.flights TO matt;
```

运行此查询的用户将在 hive catalog 下 ontime schema 中 flights 表的 SELECT 权限授予用户 matt。

你可以选择使用 WITH GRANT OPTION 来允许被授权者 matt 授予其他对象相同的权限；你也可以通过逗号分隔来指定一个以上的权限，或者使用 ALL PRIVILEGES 来授予对象的 SELECT、INSERT 和 DELETE 权限。

```
GRANT SELECT, DELETE on hive.ontime.flights TO matt WITH GRANT OPTION;
```



要授予权限，你必须拥有相同的权限和 GRANT OPTION，或者你必须是表或视图的所有者，抑或是拥有表或视图的角色成员。在撰写本书时，Presto 没有办法改变对象的所有者，所以这必须由底层数据源来完成。例如，你可以运行下面的 SQL 语句：

```
ALTER SCHEMA ontime SET OWNER USER matt;
```

角色由一组可以分配给用户或另一个角色的权限所组成，它使得管理多个用户的权限变得更加容易。通过使用角色，你可以避免直接将权限分配给用户。相反，你把权限分配给一个角色，然后将用户分配给该角色并使其继承这些权限。多个角色也可以分配给一个用户。一般来说，使用角色来管理权限是最佳实践。

让我们再次使用 flights 表的例子，并使用角色：

```
CREATE ROLE admin;
GRANT SELECT, DELETE on hive.ontime.flights TO admin;
GRANT admin TO USER matt, martin;
```

现在，假设你想移除一个用户的权限。你可以简便地从用户身上移除该角色，而不是逐个去移除该用户对某一对象的所有权限：

```
REVOKE admin FROM USER matt;
```

除了移除分配给用户的角色之外，你还可以从角色中移除权限，使该角色的所有用户不再拥有该权限：

```
REVOKE DELETE on hive.ontime.flights FROM admin;
```

在这个例子中，我们撤销了 admin 角色在 flights 表上的 DELETE 权限。但是，admin 角色及其成员仍然拥有 SELECT 权限。

用户可能属于多个角色，而这些角色可能有独立或交叉的权限集合。当用户运行一个查询时，Presto 会检查用户自己的权限和通过角色所分配到的权限。如果你只想使用自己所属的单个角色的权限，则可以使用 SET ROLE 命令。假设你同时属于 admin 角色和 developer 角色，但只想使用分配给 developer 角色的权限：

```
SET ROLE developer;
```

也可以将角色设置为 ALL，这时 Presto 会检查每个角色的权限；当然，还可以将其设置为 NONE。

10.3 加密

加密是将数据从可读形式转化为不可读形式的过程，可用于传输或存储 [也称为**静态**（at rest）数据] 中。在接收端，只有经过授权的用户才能将数据转换回可读形式，这可以防止任何截获数据的恶意攻击者读取数据。Presto 使用标准的加密技术对传输和存储中的数据进行加密。表 10-1 对比了明文数据和加密数据。

表10-1：等效明文格式和加密格式的比较

明　　文	密　　文
SSN: 123-45-6789	5oMgKBe38tSs0pl/Rg7lITExIWtCITEzIfSVydAHF8Gux1cpnCg=

传输中的加密数据存在于以下数据传输链路，如图 10-3 所示。

- 客户端（如 JDBC 客户端或 Presto CLI）和 Presto 协调器之间（图 10-3 左侧）。
- Presto 集群内，协调器与工作节点之间（图 10-3 中心）。
- 从 catalog 中配置的数据源到集群中的工作节点和协调器（图 10-3 右侧）。

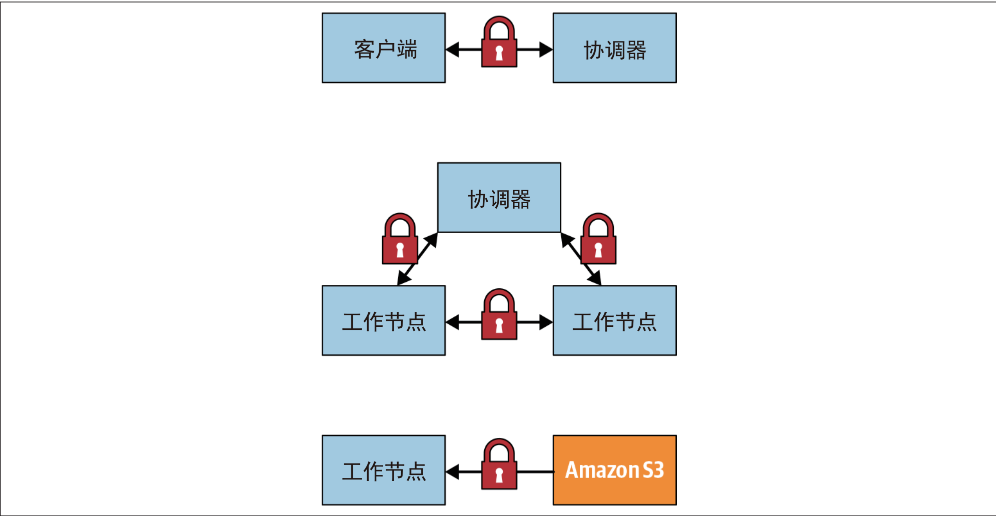


图 10-3：数据传输中的加密场景

对静态数据的加密包括以下几个地方，如图 10-4 所示。

- Presto 集群之外的数据源（图 10-4 右侧）。

- Presto 集群内工作节点和协调器上用于磁盘溢出功能的存储（图 10-4 左侧）。

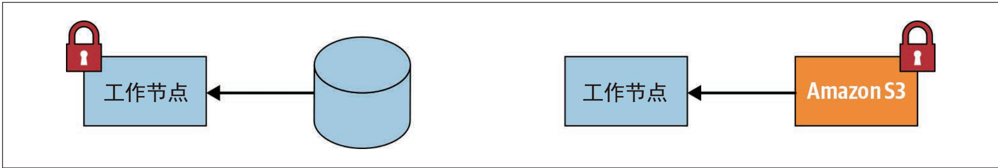


图 10-4：静态数据的加密场景

上述的每一个组件都可以在 Presto 中独立配置，图 10-4 中的 S3 只是连接数据源的一个例子。

这些不同的加密用法可以组合在一起。例如，你可以将 Presto 配置为加密客户端与协调器的通信以及集群间通信，但不对静态数据加密，因此不安全；或者可以选择配置 Presto 只加密客户端与协调器的通信。

虽然每个组合都可以配置，但有些组合并没有什么意义。例如，只加密集群间通信而不加密客户端与协调器之间的通信是没有意义的，因为这样做会导致 Presto 客户端查询时所访问到的任何数据都可能受到攻击。

如前所述，Presto 中的外部和内部通信完全通过 HTTP 进行。为了确保客户端和协调器之间的通信以及集群间的通信安全，Presto 可以配置为在 HTTP 之上使用传输层安全（TLS），简称 HTTPS。TLS 是一种在网络上加密数据的加密协议，HTTPS 使用 TLS 来保护 HTTP 的安全。



TLS 是安全套接字层（SSL）的继承者，有时这两个术语可以互换使用。SSL 是一个较老的协议，存在着已知的漏洞，被认为是不安全的。因为 SSL 很有知名度并且其名称已得到广泛认可，所以当有人提到 SSL 时，往往也是指 TLS。

日常上网的你可能对 HTTPS 并不陌生，因为现在大多数网站使用 HTTPS。如果你登录网上银行账户，HTTPS 就用于加密 Web 服务器和你的 Web 浏览器之间的数据。在现代 Web 浏览器上，你通常会在地址行中看到一个挂锁图标，表明数据传输是安全的，而且你所连接的服务器与证书中标识的服务器一致。

HTTPS 是如何工作的

虽然本章并不打算深入探讨 TLS 的技术细节，但你应该了解基本概念，以便理解 HTTPS 如何与 Presto 一起工作。我们已经讨论了 Presto 如何通过 HTTPS 来使用 TLS 加密传输中的数据。任何恶意用户在网络上窃听和拦截 Presto 中正在处理的数据，都将无法查看原始的未加密数据。

当用户从浏览器连接到网页时，网页所在的服务器发送 TLS 证书以启动 TLS 握手，就这样开启了创建安全通信的过程。TLS 证书依赖于公钥（非对称）密码学，在握手过程中，它被用来建立一个用于会话期间对称加密的密钥。在握手过程中，双方协商要使用的加密算法，并验证 TLS 证书是否可信，以及服务器是否和证书声明的一致。在握手后，双方生成会话密钥，用于加密数据。握手完成后，双方之间的数据仍是加密的，这些数据将在会话期间建立的安全信道中传输，如图 10-5 所示。

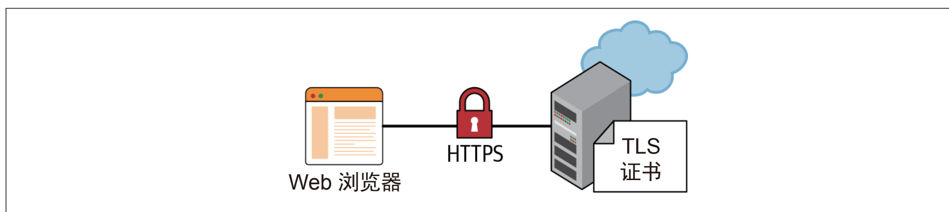


图 10-5：使用 TLS 来保护客户端 Web 浏览器和 Web 服务器之间的通信

10.3.1 加密Presto客户端与协调器之间的通信

保证客户端和 Presto 之间的流量安全很重要，原因有二：首先，如果你使用的是 LDAP 认证，则密码是以明文形式传输的，而使用 Kerberos 认证，SPNEGO 令牌也可以被拦截；其次，查询返回的任何数据都是明文的。

了解加密算法 TLS 握手的底层细节对于理解 Presto 如何加密网络流量并不重要，但了解更多关于证书的信息很重要，因为你需要创建和配置证书以供 Presto 使用。图 10-5 描述了客户端 Web 浏览器和 Web 服务器之间通过 HTTPS 加密的通信。这正是 Presto 协调器和 Presto 客户端之间的 HTTPS 通信，如 Presto Web UI、Presto CLI 或 JDBC 驱动，如图 10-6 所示。

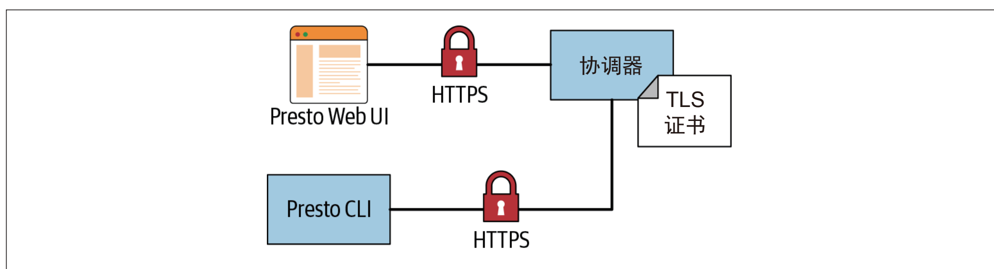


图 10-6：在 Presto 客户端和 Presto 协调器之间通过 HTTPS 进行安全通信

TLS 证书依赖于使用密钥对的公钥（非对称）密码学。

- 公钥：任何人都可以获得。

- 私钥：由所有者私有。

任何人都可以使用公钥对信息进行加密，而只有拥有私钥的人才能解密。因此，只要密钥对的所有者不分享或丢失私钥，任何用公钥加密的消息都应该是秘密的。TLS 证书包含证书的域、证书的签发人或公司、公钥以及其他几项信息，这些信息用私钥进行散列和加密。签名证书是在证书中创建签名的过程。

这些证书通常由 DigiCert 或 GlobalSign 等可信的证书机构签发。这些权威机构会验证申请颁发证书的人是否是他们所说的人，以及他们是否拥有证书中所述的域。证书使用权威机构的私钥签名，它们的公钥很容易获取到，通常默认安装在大多数操作系统和 Web 浏览器上。

在 TLS 握手过程中，证书的签名过程是验证真实性的重要环节。客户端使用公钥对来解密签名，并与证书中的内容进行比较，以确保其没有被篡改。

现在你了解了 TLS 的基础知识，让我们来看看如何在 Presto 客户端和协调器之间加密数据。为了在 Presto 协调器上启用 HTTPS，需要在 config.properties 文件中设置额外的属性（见表 10-2）。

表10-2：用于HTTPS通信的配置属性

属　性	描　述
`http-server.https.enabled`	设置为 `true` 开启 Presto 的 HTTPS，默认为 `false`
`http-server.http.enabled`	设置为 `false` 关闭 Presto 的 HTTP，默认为 `true`
`http-server.https.port`	设置 HTTPS 所使用的端口，8443 是 Java 应用服务器的默认端口
`http-server.https.keystore.path`	指定 Java keystore 文件的路径，该文件包含了 Presto 使用 TLS 时要用到的私钥和证书
`http-server.https.keystore.key`	指定访问 Java keystore 时的密码



即使你配置 Presto 使用 HTTPS，默认情况下 HTTP 也是启用的，启用 HTTPS 并不会禁用 HTTP。如果你想禁用 HTTP 访问，就需要对 HTTP 进行配置。然而，有时你可能希望在完成 Presto 安全环境的配置之前，一直启用 HTTP。这是调试问题的好方法，因为这样的话，如果在配置过程中遇到了复杂的问题，你可以通过 HTTP 测试一些东西是否在工作或如何工作。

作为一个例子，你可以将下面这几行添加到 config.properties 中：

```
http-server.https.enabled=true
http-server.https.port=8443
http-server.https.keystore.path=/etc/presto/presto_keystore.jks
http-server.https.keystore.key=slickpassword
```

记得在更新属性文件后要重新启动 Presto 协调器。

Java keystore 和 Java truststore

在配置加密的同时，你需要使用 Java keystore。keystore 是用于存储加密密钥、X.509 证书链和信任证书的存储库。在 Presto 中，keystore 也可以当作 truststore 使用。这些存储的实现或使用这些存储的工具并没有什么区别，主要的区别在于 keystore 和 truststore 中存储的内容以及使用方式。keystore 用于证明你自己的身份，而 truststore 则用于确认另一个身份。

对于 Presto 来说，Presto 协调器需要一个包含私钥和签名证书的 keystore。当 Presto 客户端连接时，Presto 会将此证书作为 TLS 握手的一部分呈现给客户端。

Presto 客户端（如 CLI）使用一个 truststore，其中包含验证服务器提供的证书的真实性所需的证书。如果 Presto 协调器证书由 CA 签发，那么 truststore 包含根证书和中间证书。如果你不使用 CA，则 truststore 需要包含相同的 Presto TLS 证书。自签名证书简单来说就是证书由证书中公钥对应的私钥签名。这些证书的安全性要低得多，因为攻击者可以在中间人攻击中欺骗自签名证书。在使用自签名证书时，你必须确保客户端所连接的网络和计算机的安全性。由于整个集群通常位于一个安全的网络内，因此使用自签名证书可以保证集群间通信的安全性。

什么是 X.509 证书？

X.509 标准定义了公钥证书的格式，比如我们要生成的 Presto 安全证书。该证书包括所签发的域名、所签发的个人所在的组织、签发日期和到期日、公钥和签名。证书可以由 CA 签发，也可以自行签发。更多信息，可以参考官方标准。

10.3.2 创建Java keystore和Java truststore

Java 中的 `keytool` 是用于创建和管理 keystore 和 truststore 的命令行工具，作为 Java 开发工具包（JDK）的一部分供用户使用。让我们通过一个简单的例子来创建 Java keystore 和 Java truststore。为简单起见，我们使用自签名证书。

首先，让我们创建 Presto 协调器使用的 keystore。下面的 keytool 命令创建了一个公 / 私钥对，并将公钥封装在一个自签名的证书中。

```
$ keytool -genkeypair \
    -alias presto_server \
    -dname CN=*.example.com \
    -validity 10000 -keyalg RSA -keysize 2048 \
    -keystore keystore.jks \
    -keypass password
    -storepass password
```

生成的 keystore.jks 文件需要在服务器上使用，并在 `http-server.https.keystore.path` 属性中指定。类似的用法也适用于 `http-server.https.keystore.keystore.key` 属性中的 `storepass` 密码。

上述例子使用的是**通配符证书**，我们指定通用名（CN）为 `*.example.com`。若 Presto 集群中的所有节点使用相同的域，则这个证书可以为它们所共享。这个证书还可以和 `coordinator.example.com`、`worker1.example.com`、`worker2.example.com` 等一起使用。这种方法的缺点是 example.com 域下的任何节点都可以使用该证书。

你可以通过使用主题替代名（`SubjectAltName`）来限制子域，在这里你可以列出子域。这允许你创建单个证书，它可以为一个有限的、特定的主机列表所共享。另一种方法是为每个节点创建一个证书，这要求你为每个节点明确定义完整的域。这增加了管理负担，并且在扩展 Presto 集群时具有挑战性，因为新节点需要绑定到完整域的证书。

当客户端连接到协调器时，协调器将其证书发送给客户端以验证其真实性：如果是自签名的，则包含协调器的证书；如果是由 CA 签发的，则包含一个证书链。稍后将讨论如何使用 CA 的证书链。因为 keystore 也包含了证书，所以你可以简单地将 keystore 复制到客户端机器上，并将其作为 truststore 使用。然而，这并不安全，因为 keystore 也包含了我们想要保密的私钥。要创建自定义的 truststore，你需要从 keystore 中导出证书，然后将其导入 truststore 中。

首先，在你创建 keystore 的协调器上导出证书：

```
$ keytool --exportcert \
    -alias presto_server \
    -file presto_server.cer \
    -keystore keystore.jks \
    -storepass password
```

这个命令可以创建一个 presto_server.cer 证书文件。下一步可以使用该文件来创建 truststore：

```
$ keytool --importcert \
    -alias presto_server \
    -file presto_server.cer \
    -keystore truststore.jks \
    -storepass password
```

因为证书是自签名的，所以这个 `keytool` 命令会提示你确认信任该证书。只需键入 **`yes`**，truststore.jks 就会被创建。现在，你可以安全地将这个 truststore 分发到任何使用客户端连接协调器的机器上了。

我们已经使用 keystore 的 HTTPS 启用了协调器，并且为客户端创建了 truststore，现在可以安全地连接到 Presto，并且客户端和协调器之间的通信是加密的。下面是使用 Presto CLI 的例子：

```
$ presto --server https://presto-coordinator.example.com:8443 \
      --truststore-path ~/truststore.jks \
      --truststore-password password
```

10.3.3 在Presto集群内加密通信

接下来看一下如何再次使用 HTTP over TLS 来保障 Presto 集群内工作节点之间以及工作节点和协调器之间的通信，如图 10-7 所示。

虽然客户端与协调器之间的通信可能是通过不受信任的网络进行的，但 Presto 集群通常部署在一个更安全的网络上，这使得集群内的安全通信不是必需的。不过，如果担心恶意攻击者能够进入集群的网络，则可以对通信加密。

与确保客户端与协调器之间的通信安全一样，集群内部通信也依赖于同一个 keystore。你在协调器上创建的这个 keystore 必须分发给所有的工作节点。

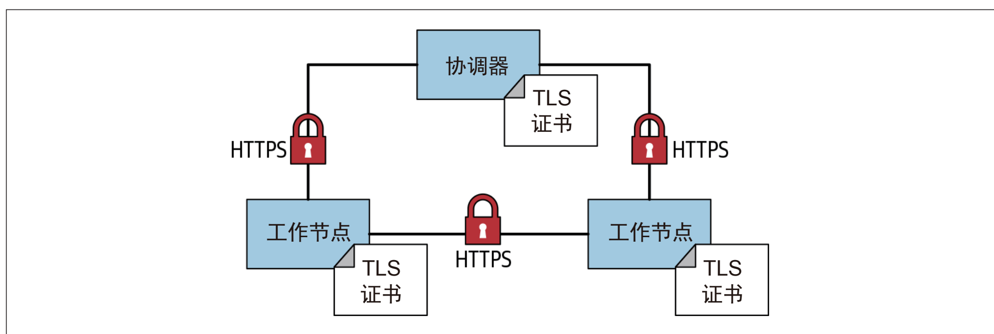


图 10-7：Presto 集群中节点之间通过 HTTPS 进行安全通信

执行 TLS 握手从而在客户端和服务器之间建立信任和创建加密通道的方法，也适用于集群间通信。集群中的通信是双向的，即一个节点可以作为客户端向另一个节点发送 HTTPS 请求，而当节点接收到请求并将证书呈现给客户端进行验证时，它也可以作为服务器。由于面向不同的连接，每个节点同时充当客户端和服务器，因此它需要一个同时包含私钥和证书中公钥的 keystore。

所有节点都需要在 config.properties 中启用 HTTPS，包括内部通信：

```
http-server.https.enabled=true
http-server.https.port=8443

internal-communication.https.required=true
discovery.uri=https://coordinator.example.com:8443

internal-communication.https.keystore.path=/etc/presto/presto_keystore.jks
internal-communication.https.keystore.key=slickpassword
```

别忘了在更新属性文件后重启工作节点。现在你已经对内部和外部的通信做出了完全的保护，并可以防止网络上的窃听者从 Presto 截获数据。



一旦一切都已正常工作，记得在 config.properties 中设置 `http-server.http.enabled=false` 来禁用 HTTP；否则，用户仍可以使用 HTTP 连接到集群。

10.4 CA与自签名证书

当你第一次尝试 Presto 并努力将其配置得更安全时，最容易的方式是使用自签名证书。然而实践中，你的组织可能不允许使用它，因为在某些情况下它们不是很安全，并且容易受到攻击。因此，你可以使用由 CA 数字签名的证书。

一旦你创建了 keystore，就需要创建一个证书签名请求（CSR）以发送给 CA 对 keystore 签名。CA 会验证你是否是你说的那个人，并向你发出由他们签名的证书。然后，该证书被导入你的 keystore 中，并将提供给客户端来替代之前的自签名证书。

一个有趣的地方与 Java truststore 的使用有关。Java 提供了一个默认的 truststore，其中可能已经包含了 CA。在这种情况下，提交给客户端的证书可以通过默认的 truststore 来验证，而在这个默认 truststore 不包含 CA 时就可能很麻烦。或者你的组织可能有自己的内部 CA，用于向员工和服务发布组织证书。因此，即使你使用了 CA，仍建议你创建自己的 truststore，以供 Presto 使用。不过，此时你可以导入 CA 证书链，而不是 Presto 使用的实际证书。**证书链**是由两个或多个 TLS 证书组成的列表，其中证书链中的每个证书都由链中的下一个证书签名。在链的顶端是根证书，这总是由 CA 自己签名。它被用于签发下游的证书——**中间证书**。你发布的 Presto 证书是由中间证书签名的，也就是链中的第一个证书。这样做的好处是，如果有多个 Presto 集群的多个证书或证书被重新签发，无须每次都把它们重新导入你的 truststore 中，这减少了 CA 重新签发中间证书或根证书的需求。

图 10-8 中的场景展示了 CA 签发的证书的用法。truststore 只包含 CA 的中间证书和根证书。来自 Presto 协调器的 TLS 证书在客户端 truststore 中使用该证书链进行验证。

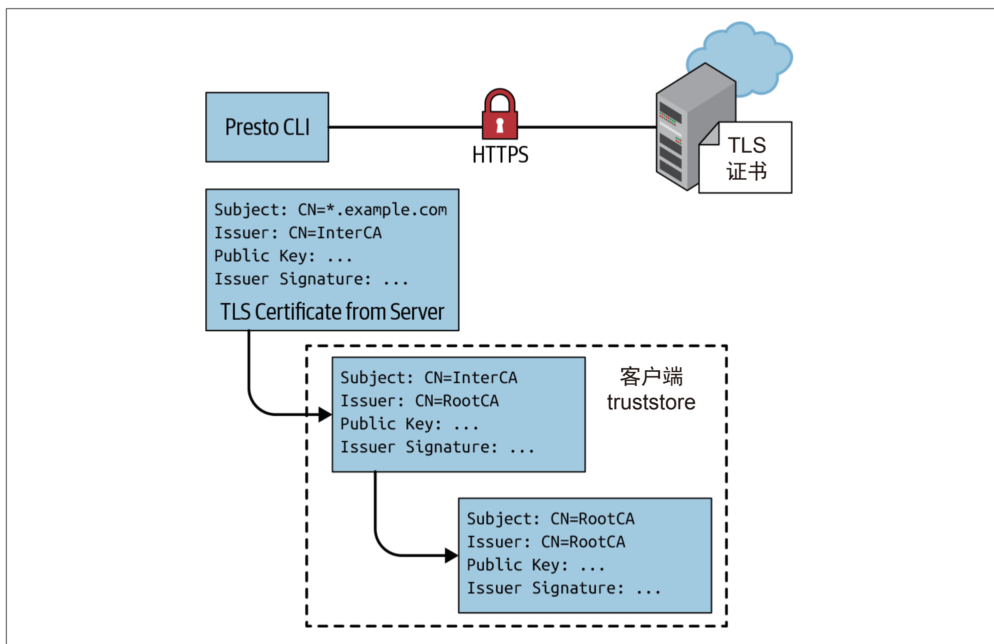


图 10-8：Presto 使用由 CA 签发的证书

假定你的 Presto 证书是由 CA 签发的。为了让客户端信任它，我们需要创建一个包含中间证书和根证书的 truststore。就像前面导入 Presto 自签名证书的例子一样，你也要执行同样的 CA 证书链的导入：

```
$ keytool --importcert \
    -alias presto_server \
    -file root-ca.cer \
    -keystore truststore.jks \
    -storepass password
```

在导入根 CA 证书后，继续从链上导入所有必要的中间证书：

```
$ keytool --importcert \
    -alias presto_server \
    -file intermediate-ca.cer \
    -keystore truststore.jks \
    -storepass password
```

注意，中间证书可能不止一个，为了简单起见，这里只使用一个。

10.5 证书认证

现在你已经了解了 TLS、证书和 Java `keytool` 的相关用法，下面可以看看如何利用这些工具对使用 TLS 连接到 Presto 的客户端进行验证。这种证书认证如图 10-9 所示。

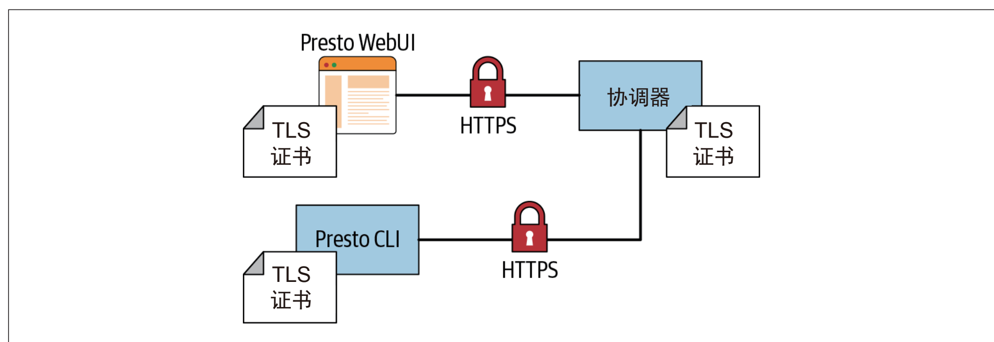


图 10-9：Presto 客户端的证书认证

作为 TLS 握手协议的一部分，服务器向客户端提供证书，以便客户端对服务器进行认证。**相互 TLS** 指在握手过程中客户端也要向服务器提供证书，以便服务器对客户端进行认证。服务器验证证书的方式与你所看到的客户端验证证书的方式相同。为了验证，服务器需要包含 CA 链的 truststore 或自签发证书。

要配置 Presto 协调器进行相互 TLS 认证，你需要在协调器的 config.properties 文件中添加一些属性。让我们来看一个完整的配置：

```
http-server.http.enabled=false
http-server.https.enabled=true
http-server.https.port=8443

http-server.https.keystore.path=/etc/presto/presto_keystore.jks
http-server.https.keystore.key=slickpassword

http-server.https.truststore.path=/etc/presto/presto_truststore.jks
http-server.https.truststore.key=slickpassword

node.internal-address-source=FQDN
internal-communication.https.required=true
internal-communication.https.keystore.path=/etc/presto/presto_keystore.jks
internal-communication.https.keystore.key=slickpassword

http-server.authentication.type=CERTIFICATE
```

属性 `http-server.authentication` 表示要使用的认证类型。在这种情况下，Presto 使用的是 `CERTIFICATE` 认证，这使得 Presto 协调器使用完整的 TLS 握手协议进行相互认证。尤其是，服务器端的协调器将证书请求消息作为完整的 TLS 握手协议的一部分发送给客户端，以提供已签名的证书用于验证。此外，为了验证客户端提交的证书，你需要在协调器上配置 truststore。

下面用 CLI 命令连接到 Presto：

```
$ presto --server https://presto-coordinator.example.com:8443 \
      --truststore-path ~/truststore.jks \
      --truststore-password password
      --user matt
presto> SELECT * FROM system.runtime.nodes;
Error running command: Authentication failed: Unauthorized
```

你会发现认证失败了，因为客户端没有使用拥有证书的 keystore 来向协调器提供客户端证书进行相互认证。

你需要修改命令来包含 keystore。注意，这个 keystore 与集群上的 keystore 不同，这个 keystore 专门包含了客户端的密钥对。下面先在客户端上创建 keystore。

```
$ keytool -genkeypair \
    -alias presto_server \
    -dname CN=matt \
    -validity 10000 -keyalg RSA -keysize 2048 \
    -keystore client-keystore.jks \
    -keypass password
    -storepass password
```

在这个例子中，可以看到我们将 CN 设置为用户 matt。在这种情况下，这很有可能是一个自签发证书，或者是所在组织有自己的内部 CA。下面在 CLI 命令中指定客户端 keystore：

```
$ presto --server https://presto-coordinator.example.com:8443 \
      --truststore-path ~/truststore.jks \
      --truststore-password password
      --keystore-path ~/client-keystore.jks \
      --keystore-password password
      --user matt

presto> SELECT * FROM system.runtime.nodes;
Query failed: Access Denied:
Authenticated user AuthenticatedUser[username=CN=matt,principal=CN=matt]
cannot become user matt
```

现在我们已经认证了，但授权失败了。回顾一下，认证是证明你是谁，而授权则控制你能做什么。

对于证书认证，Presto 会从 X.509 证书中提取主体的专有名称。这个值作为安全主体与用户名进行比较。除非在 CLI 中使用 `---user` 选项明确指定，否则用户名默认为操作系统的用户名。本例中，用户 matt 会与证书 `CN=matt` 中的可分辨名称比较。一种变通方法是将该选项作为 `--user CN=matt` 传递给 CLI。另外，你也可以利用前面学过的内置的、基于文件的系统访问控制进行一些自定义。

首先，需要在 Presto 协调器上的 Presto 安装目录创建一个文件 access-control.properties：

```
access-control.name=file
security.config-file=/etc/presto/rules.json
```

然后需要在协调器上创建 rules.json 文件，放在 access-control.properties 文件中指定的路径位置，并定义从 `principal` 到 `user` 的映射，其中必须包含 `CN=`：

```
{
  "catalogs": [
    {
      "allow": true
    }
  ],
  "principals": [
    {
      "principal": "CN=(.*)",
      "principal_to_user": "$1",
      "allow": true
    }
  ]
}
```

我们正将一个安全主体的正则表达式与捕获组进行匹配。然后，我们使用这个捕获组来将安全主体映射到用户。在我们的例子中，regex 匹配 `CN=matt`，其中 `matt` 是捕获组中的一部分，它被映射到用户。当你创建了这些文件并重启协调器后，证书认证和用户对应的安全主体认证都会生效：

```
SELECT * FROM system.runtime.nodes;
-[ RECORD 1 ]+--------------------------------------------------------
node_id      | i-0779df73d79748087
http_uri     | https://coordinator.example.com:8443
node_version | 312
coordinator  | true
state        | active
-[ RECORD 2 ]+--------------------------------------------------------
node_id      | i-0d3fba6fcba08ddfe
http_uri     | https://worker-1.example.com:8443
node_version | 312
coordinator  | false
state        | active
```

10.6 Kerberos

网络认证协议 Kerberos 使用广泛。Presto 支持 Kerberos 对使用 Hive 连接器的 Presto 用户来说尤为关键（参见 6.4 节），因为 Kerberos 是 HDFS 和 Hive 常用的验证机制。



Kerberos 文档对学习协议以及相关概念和术语非常有用。本节假设你对这些方面已足够熟悉，或者已阅读了一些文档和其他资料。

Presto 支持客户端通过使用 Kerberos 验证机制在协调器那里验证身份。Hive 连接器可以在使用 Kerberos 认证的 Hadoop 集群那里验证身份。

与 LDAP 类似，Kerberos 是一种认证协议，安全主体可以使用用户名和密码或 keytab 文件进行认证。

10.6.1 前提条件

Kerberos 需要在 Presto 协调器上配置，该协调器需要能够连接到 Kerberos **密钥分发中心**（key distribution center，KDC）。KDC 负责认证安全主体，并签发可以与已启用 Kerberos 的服务一起使用的会话密钥。KDC 通常使用 TCP/IP 端口 88。

要使用 MIT Kerberos，你需要在 /etc/krb5.conf 配置文件中设置一个 `[realms]` 段。

10.6.2 Kerberos客户端认证

要启用 Presto 的 Kerberos 认证，你需要在 Presto 协调器的 config.properties 文件中添加详细信息；还需要更改验证类型，配置 keytab 文件的位置，并指定要使用的 Kerberos 服务账户的用户名：

```
http-server.authentication.type=KERBEROS
http.server.authentication.krb5.service-name=presto
http.server.authentication.krb5.keytab=/etc/presto/presto.keytab
```

对于客户端的认证，不需要对工作节点配置进行任何更改。工作节点继续通过未认证的 HTTP 连接到协调器。

要连接到这个已开启 Kerberos 的 Presto 集群，用户需要在客户端上设置他们的 keytab 文件、安全主体和 krb5.conf 配置文件，然后使用相关的 Presto CLI 参数或 JDBC 连接的属性。你可以在 Presto 文档中找到所有的细节，其中还包括了一个小的封装脚本。

10.6.3 集群内部Kerberos

如果你想确保集群内部通信的安全，必须在工作节点上启用 Kerberos 认证，并且需要将内部通信更改为使用 SSL/TLS，参见 10.3.3 节。这需要为内部通信指定有效的 Kerberos 凭证。

Presto 本身，以及任何用 Kerberos 连接到 Presto 的用户，都需要 Kerberos 安全主体。你需要用 `kadmin` 在 Kerberos 中创建这些用户。此外，Presto 协调器还需要 keytab 文件。

当使用 Kerberos 认证时，客户端应使用 HTTPS 访问 Presto 协调器，参见 10.3.1 节。

如果你想让 Presto 在 Kerberos 安全主体使用某个 host 值而不是机器的主机名，则可以设置协调器的 Kerberos 主机名（这是个可选设置）。你还可以指定 Kerberos 配置文件 krb5.conf

的所在位置，而不是默认的 /etc/krb5.conf：

```
http.server.authentication.krb5.principal-hostname=presto.example.com
http.authentication.krb5.config=/etc/presto/krb5.conf
```

为了保证集群内部通信的安全，你需要为内部通信指定有效的 Kerberos 凭证并启用它：

```
internal-communication.kerberos.enabled=true
```

确保你也在工作节点上设置了 Kerberos。用于内部通信的 Kerberos 安全主体是由 `http.server.authentication.krb5.service-name` 建立的，其后追加了 Presto 运行节点的主机名和 Kerberos 配置的默认 realm。

10.7 数据源访问和安全配置

在图 10-10 中可以看到 Presto 数据安全的另一个方面。在 Presto 中配置的每个 catalog 都包括连接字符串以及用于连接到数据源的用户凭证。不同的连接器和目标数据源系统允许不同的访问配置。

用户首先向协调器进行认证。Presto 连接器向数据源发出请求，这通常也需要认证。

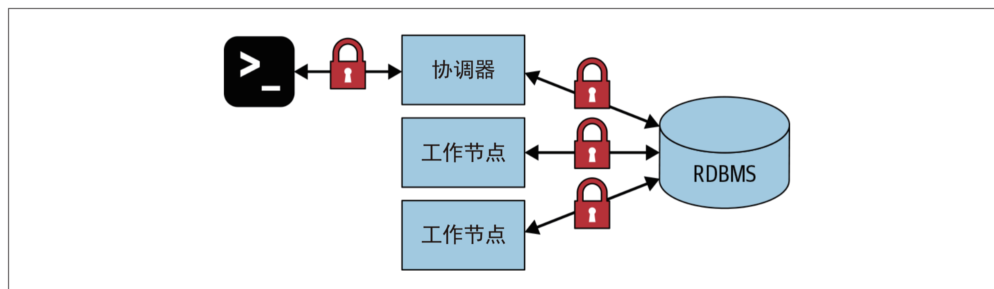


图 10-10：影响用户数据的数据源安全配置

从连接器到数据源的认证取决于连接器的实现。在许多连接器实现中，Presto 以服务用户的身份进行认证。因此，对于这类连接器来说，无论执行查询的用户是谁，查询都是以服务用户的身份在底层系统中执行的。

如果连接目标数据源使用的用户凭证不包括任何写权限，那么你可以有效地限制所有的 Presto 用户只能进行只读操作和查询。同样，如果该用户凭证没有对特定 schema、数据库甚至表的访问权限，使用 Presto 查询也有同样的限制。

为了提供更精细的访问控制，你可以创建多个具有不同权限的服务用户。这样就可以用相同的连接器连接到同一个数据源来配置多个 catalog，但使用不同的服务用户。

与使用不同的服务用户类似，你也可以用不同的连接字符串创建 catalog。例如，PostgreSQL 连接字符串包括一个数据库名称，这意味着你可以创建不同的 catalog，从而分开访问这些运行在同一 PostgreSQL 服务器上的数据库。类似地，MS SQL Server 连接字符串允许对可选的配置指定数据库。

具体连接字符串、用户凭证等问题已在第 6 章讨论过。

除了 schema 和 database 层面，你甚至可以将访问权限和数据内容配置一直下推到数据库本身。如果你想限制对表中的某些列或表中的某些行的访问权限，则可以限制对源表的访问权限，并创建具有所需内容的视图。这些视图在 Presto 中就像使用传统表一样，从而实现了安全性。极端的场景是创建单独的数据库或使用 ETL 工具的数据仓库，甚至包括 Presto 本身。可以在 Presto 中单独为这些带有所需数据的目标数据库配置 catalog 访问。

如果你最终在 Presto 部署中用前面的一些逻辑定义了多个 catalog，并且想允许特定用户访问这些 catalog，则可以利用系统访问控制来做到这一点（参见 10.2.1 节）。

在一些 Presto 连接器和商业连接器中，一个相对较新的功能是终端用户扮演（end-user impersonation）。这允许将 Presto CLI 或其他工具中的终端用户凭证一路传递到数据源，之后在 Presto 中的访问权限就可以反映数据库中配置的访问权限。

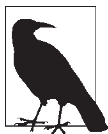
数据源安全配置的一个常见例子是使用 Kerberos 连接 HDFS 和使用 Hive 连接器（参见 6.4 节），我们将在 10.8 节中介绍相关细节。

10.8 使用Hive连接器进行Kerberos验证

了解了 Kerberos 配置和数据源安全的基础知识后，下面来看看 Kerberos、HDFS/Hive 和 Hive 连接器的组合使用。

默认情况下，Hive 连接器没有启用认证，但该连接器支持 Kerberos 认证。你需要做的就是将连接器配置为与 Hadoop 集群上的两个服务一起工作。

- Hive Metastore Thrift 服务
- HDFS



如果你的 krb5.conf 不在 /etc/krb5.conf 内，则必须使用 jvm.config 文件中的 `java.security.krb5.conf` 这个 JVM 属性显式地进行设置。

```
-Djava.security.krb5.conf=/example/path/krb5.conf
```

10.8.1 Hive Metastore Thrift服务认证

在开启了 Kerberos 的 Hadoop 集群中，Presto 通过使用**简单认证和安全层**（simple authentication and security layer，SASL）连接到 Hive Metastore Thrift 服务，并使用 Kerberos 进行认证。你可以在 catalog 属性文件中轻松地为 HMS 启用 Kerberos 认证。

```
hive.metastore.authentication.type=KERBEROS
hive.metastore.service.principal=hive/hive-metastore-host.example.com@EXAMPLE.COM
hive.metastore.client.principal=presto@EXAMPLE.COM
hive.metastore.client.keytab=/etc/presto/hive.keytab
```

这个设置激活了将 Kerberos 用于认证 HMS 的功能，它还配置了连接到 Metastore 服务时 Presto 所使用的 Kerberos 安全主体和 keytab 文件的位置。注意，keytab 文件必须分发到集群中的每个节点。



`hive.metastore.service.principal` 可以在其值中使用 `_HOST` 占位符。当连接到 HMS 时，Hive 连接器会用它所连接的 Metastore 服务器的主机名来代替这个值。如果 Metastore 在多个主机上运行，这会很有用。同样，`hive.metastore.client.principal` 也可以在其值中使用 `_HOST` 占位符。当连接到 HMS 时，Hive 连接器会用 Presto 工作节点的主机名来代替。如果每个工作节点都有自己的 Kerberos 安全主体，则这种配置方式非常有用。

Presto 以 `hive.metastore.client.principal` 属性指定的 Kerberos 安全主体的身份发起连接，并使用 `hive.metastore.client.keytab` 属性指定的 keytab 来验证这个安全主体。它验证 Metastore 的身份是否与 `hive.metastore.service.principal` 匹配。



由 `hive.metastore.client.principal` 指定的安全主体必须有足够的权限来删除 hive/war☺ouse 目录内的文件和子目录。如果没有这个权限，则只有元数据会被删除，而数据本身会继续消耗磁盘空间，这是因为由 HMS 负责删除内部表数据。当 Metastore 被配置为使用 Kerberos 认证时，由 Metastore 执行的所有 HDFS 操作都会由 Presto 代办。删除数据的错误会被默默忽略。

10.8.2 HDFS认证

对 HDFS 启用 Kerberos 认证类似于 Metastore 的认证，在连接器属性文件中需要设置的属性如下所示。当验证类型为 `KERBEROS` 时，Presto 会以 `hive.hdfs.presto.principal` 属性指定的安全主体来访问 HDFS。Presto 使用由 `hive.hdfs.presto.presto.keytab` 属性指定的 keytab 来验证这个安全主体。

```
hive.hdfs.authentication.type=KERBEROS
hive.hdfs.presto.principal=hdfs@EXAMPLE.COM
hive.hdfs.presto.keytab=/etc/presto/hdfs.keytab
```

10.9 集群分离

另一个大规模的安全方案是，通过在相互独立的 Presto 集群上使用数据源和配置 catalog，实现它们的完全分离。这种分离对很多场景非常有用。

- 将只读操作从 ETL 和其他写操作的场景中分离出来。
- 由于数据监管要求，需要在完全不同的基础设施上托管集群。例如，一边是 Web 流量数据，一边是医疗、金融或个人数据等受严格监管的数据。

这种集群的分离可以让你针对不同的场景和数据优化集群配置，或者将集群放置在更接近数据的地方。这两种情况都可以在满足安全需求的同时获得巨大的性能和成本优势。

10.10 小结

现在，你可以对你的数据和 Presto 提供的数据访问能力感到放心了。你知道许多用来确保 Presto 访问与数据安全的选项。这是运行 Presto 的一个关键部分，并且必须通过其他活动（如监控）来加强，第 12 章将对此进行讨论。

但在这之前，你需要先了解一下众多使用 Presto 的工具，它们的强大能力令人惊讶，参见第 11 章。

第 11 章
将Presto与其他工具集成

如第 1 章所述，你可以用各种方式使用 Presto。现在你已经学习了如何运行 Presto 集群、使用 JDBC 连接，以及针对一个或多个 catalog 编写查询。

是时候看看众多组织中 Presto 的成功应用了。下面的章节涵盖了各种应用场景，代表运用 Presto 所创造的各种可能性中的一小部分。

11.1 使用Apache Superset进行查询、可视化和更多操作

Apache Superset 可以说是一个现代的、企业级的商业智能 Web 应用程序，但这种简明扼要的描述根本不能表达出 Superset 的强大。

Superset 作为 Web 应用程序运行在基础设施中，因此业务分析师和其他用户无须在他们的工作站上安装或管理其工具。它支持 Presto 作为数据源，因此可以作为访问 Presto 和所有已配置的数据源的前端。

一旦连接到 Superset，用户就可以在功能丰富的 SQL 查询编辑器 SQL Lab 中编写查询。SQL Lab 允许用户在多个选项卡中编写查询、浏览数据库的元数据作为辅助、运行查询、并在用户界面中以表格形式接收查询结果，它甚至支持长时间运行的查询。此外，SQL Lab 还拥有众多的 UI 特性，可以帮助你编写查询，甚至将所需的工作减少到只需单击一两下按钮。对于用户来说，单单是 SQL Lab 就已很有价值并且功能很强大了，但这仅仅是受

益于 Superset 的第一步。SQL Lab 可以让你顺利过渡到 Superset 的真正功能：可视化数据。

Superset 所支持的可视化功能非常丰富，你可以通过查看可视化图库（参见 Apache Superset 网站）来感受一下。你可以创建所有典型的可视化图形，包括数据点图、线图、条形图或饼图等。然而，Superset 的功能更加强大，支持 3D 可视化、基于地图的数据分析等。

一旦创建了必要的查询和可视化，你就可以将其组成一个仪表板，并与组织内的其他成员分享。这让商业分析师和其他专家能够创建有用的数据聚合和可视化，并方便地将其暴露给其他用户。

在 Superset 中使用 Presto 很简单。在这两个系统都启动并运行后，你只需要在 Superset 中将 Presto 配置成一个数据库即可。

将 Presto 和 Superset 连接起来之后，建议把它慢慢地暴露给用户。Superset 的简单性让用户可以创建出强大但计算量超大并涉及大量数据集的查询，这对 Presto 集群的规模和配置有重大影响。在用户和使用场景方面逐步扩展使用量，可以让你随时跟踪集群的利用率，并确保你可以根据新需求来扩展集群。

11.2 使用RubiX提高性能

当你扩展 Presto 以访问大型分布式存储系统并将其暴露给许多用户和工具时，对基础设施的要求就会大大提高。计算性能需求可以通过扩展 Presto 集群本身来处理，被查询的数据源也可以进行调整，但即使引入和适配了这些优化，也会给你留下一个缺口——Presto 和数据之间的连接。

来自 Qubole 的轻量级数据缓存框架 RubiX 作为缓存层，可以放在 Presto 计算资源和数据源之间。它支持磁盘缓存和内存缓存。在查询分布式存储系统时，因为避免了数据传输和对底层数据源的重复查询，所以使用这个开源平台可以提升性能并降低成本。

RubiX 引入了一个新的协议 `rubix://`，用于指向 Hive Metastore 中表的位置。因此，它是 Hive 连接器的透明的增强，从 Presto 用户的角度来看并没有什么变化。元信息以及实际数据都缓存在 RubiX 中。RubiX 存储是分布式的，可以与 Presto 集群部署在相同节点上，以获得最大的性能提升。

将 RubiX 和 Presto 一起使用是个既定做法，因为在查询分布式对象存储时，二者的优势非常互补。

11.3 使用Apache Airflow的工作流

Apache Airflow 是一个广泛使用的系统，用于以编程方式编写、调度和监控工作流。它专注于数据流水线处理，在数据科学界得到了广泛的应用。它用 Python 实现，能够编排并执行用 Python 编写的复杂工作流，并且能够调用众多支持的系统，还可以执行 shell 脚本。

为了将 Presto 与 Airflow 集成起来，你可以利用 Airflow 中的 Presto 钩子（hook），或在命令行中运行 Presto CLI。Airflow 支持 Presto 之外的许多数据源，因此你既可以在 Presto 之外准备数据，之后通过 Presto 来消费数据，又可以通过 Presto 来访问和处理数据，以利用其高性能、可扩展性和集成众多数据源的特性。

整合 Airflow 和 Presto 的目标通常是对源数据进行处理，最终得到一个高质量的数据集，可以为应用程序和机器学习模型所用。一旦由 Airflow 编排并由 Presto 或其他集成的组件执行的工作流程产生了所需的数据集，就可以用 Presto 来访问这些数据，并通过报告和仪表板向用户展示这些数据，这部分可以使用 Apache Superset，参见 11.1 节。

11.4 嵌入式Presto示例：Amazon Athena

Amazon Athena 是一个查询服务，可用于直接对存储在 Amazon S3 中的任何大小的数据文件执行 SQL 查询。Athena 是一个很好的应用程序例子，它将 Presto 作为查询引擎封装起来，从而提供强大的功能和特性。Athena 以服务的形式提供给用户，本质上是使用 Presto 和 Hive 连接器来查询 S3。与存储一起使用的 Hive Metastore 是另一个服务，AWS Glue。

图 11-1 展示了 Amazon Athena 的总体架构。客户端通过 Athena Presto REST API 访问 Athena。通过使用 Glue Data Catalog（保存 S3 上数据的源数据）与在 Athena 上部署的 Presto 进行交互，由 Presto 执行查询。

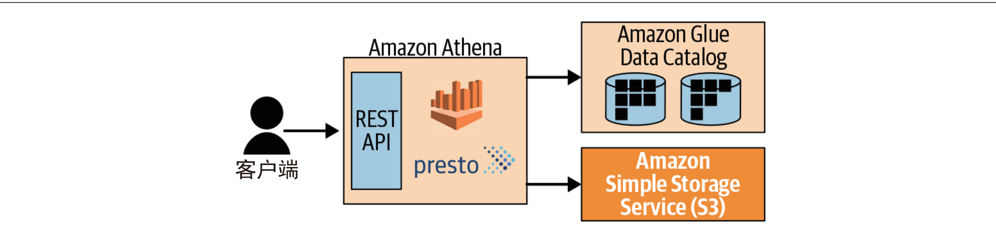


图 11-1：Amazon Athena 架构概览

Athena 是一种**无服务器架构**，这意味着你既无须管理任何基础设施［如 Amazon Elastic Compute Cloud（EC2）实例］，也无须管理、安装或升级任何数据库软件来处理 SQL 语句。Athena 为你解决了这一切。无须花费时间部署，Athena 可以立即为你提供提交 SQL 查询的端点。无服务器是 Athena 的一个关键设计，这种架构天生具备高可用性和容错性优势。亚

马逊为其持续运行和可用性提供了保证，同时还提供了对故障和数据丢失的容错性。

因为 Athena 是无服务器的，所以它使用的定价模式与你自己管理基础设施不同。例如，当在 EC2 上运行 Presto 时，无论你使用 Presto 的量如何，你都要按小时为 EC2 实例付费；而使用 Athena，你只需按每次查询从 S3 读取的数据量来支付相应费用。

Amazon 提供了多种客户端与 Athena 交互，并向 Athena（即 Presto）提交查询。你可以使用 AWS 命令行界面、REST API、AWS Web 控制台或在应用程序中使用 JDBC 驱动、ODBC 驱动或 Athena SDK。

尽管有如上诸多优点，但我们仍要明白，从用户的角度来看，Athena **并不是 Presto 托管部署**。以下是其区别于 Presto 部署的几个重要方面。

- 它并不能使用 AWS 内部或者外部其他的数据源。
- 你无法访问 Presto Web UI 来了解查询详情和其他相关信息。
- 你无法控制 Presto 本身，包括版本、配置或基础设施。

下面来看一个使用 Athena 访问鸢尾花数据集（参见 1.4.7 节）的简单例子：创建好数据库和表，并将数据导入 Athena 后，你就可以开始使用了。

可以通过使用 AWS CLI 的 Athena 命令 `start-query-execution` 来运行 Athena 查询。你需要使用以下两个参数。

`--query-string`
: 这是你要在 Athena 中执行的查询。

`--cli-input-json`
: 这是一个 JSON 结构体，它为 Athena 提供了额外的上下文。在本例中，我们在 Glue Data Catalog 中指定了存有 `iris` 表的数据库，并指定了查询结果保存的位置。



所有在 Athena 中运行的查询都会将结果写入 S3 中的某个位置。你可以在 AWS Web Console 中进行配置，也可用 Athena 的客户端工具来指定。

下面使用这个存储在 athena-input.json 中的 JSON 结构体来运行查询：

```
{
    "QueryExecutionContext": {
        "Database": "iris"
    },
    "ResultConfiguration": {
        "OutputLocation": "s3://presto-book-examples/results/"
    }
}
```

用 AWS CLI 运行 Athena 查询：

```
$ aws athena start-query-execution \
--cli-input-json file://athena-input.json \
--query-string 'SELECT species, AVG(petal_length_cm), MAX(petal_length_cm), \
  MIN(petal_length_cm) FROM iris GROUP BY species'

{
    "QueryExecutionId": "7e2a9640-04aa-4ae4-8e88-bd6fe4d9c289"
}
```

因为 Athena 是异步执行查询，所以调用 `start-query-execution` 会返回一个查询执行 ID。它可用于获取查询执行的状态，或者在查询完成后获取结果，结果将以 CSV 格式存储在 S3 中：

```
$ aws athena get-query-execution \
  --query-execution-id 7e2a9640-04aa-4ae4-8e88-bd6fe4d9c289

{
   "QueryExecution": {
    ...
       "ResultConfiguration": {
          "OutputLocation":
             "s3://...7e2a9640-04aa-4ae4-8e88-bd6fe4d9c289.csv"
       },
      ...
}

$ aws s3 cp --quiet
s3://.../7e2a9640-04aa-4ae4-8e88-bd6fe4d9c289.csv
/dev/stdout


"species","_col1","_col2","_col3"
"virginica","5.552","6.9","4.5"
"versicolor","4.26","5.1","3.0"
"setosa","1.464","1.9","1.0"
```

你也可以使用 `aws athena get-query-results` 命令以 JSON 结构的格式获取结果。另一个选择是开源的 AthenaCLI。

停下来想想这件事：在没有**任何**基础设施需要管理的情况下，你只需要用一个命令行界面就可以在 S3 中存储的数据文件上运行 SQL 查询。如果没有 Athena 和 Glue，你必须部署和管理基础设施和软件来对数据执行 SQL 查询。

而如果没有 Presto，你必须以某种方式采集并格式化数据，然后写入数据库中，之后才可以进行 SQL 处理。

Athena 和 Glue 的结合是一种令人难以置信的强大工具，而 Presto 使你可以使用标准 SQL 对 S3 上的数据进行操作。

这个简单的介绍并不能使你全面了解 Athena，但可以让你了解到 Presto 在行业内的应用情况，以及 Presto 与其他产品的区别。

使用 Athena 可以满足不同的需求，但有一些特定的限制和特性。例如，Athena 对较大的、运行时间较长的查询施加了限制。

因为你是为处理后的数据量而不是运行软件的基础设施付费，所以成本计算也有很大不同。每次运行查询时，你都要为处理的数据付费。根据你的使用模式，这可能更便宜，也可能更贵。具体来说，为了降低查询成本，你可以在 S3 中预处理数据，从而使用 Parquet 和 ORC 等格式并进行压缩。当然，预处理也是有代价的，因此你必须去尝试优化整体成本。

许多其他平台以类似的、隐蔽的方式使用 Presto，从而为用户和这些平台的提供者提供了强大的能力。如果你追求管控力和灵活性，运行自己部署的 Presto 仍是一个很棒的选择。

11.5 Starburst企业版Presto

Starburst 是 Presto 开源项目背后的商业公司，也是 Presto 项目和 Presto 软件基金会的主要赞助商。Starburst 的创始团队成员是来自 Teradata 公司的 Presto 早期贡献者，他们创办 Starburst 是为了专注于延续 Presto 在企业中的成功。Facebook 的 Presto 开源项目创始人在 2019 年加入 Starburst，Starburst 已成为 Presto 项目最大的贡献者和代码提交者之一。

Starburst 为增强版的 Presto 提供商业支持，并为其提供了更多的企业级功能，比如更多的连接器、更好的性能和其他现有连接器的改进，以及用于集群和 catalog 管理的控制台和增强的安全功能。

Starburst 企业版 Presto 支持在任何地方部署 Presto：物理服务器、虚拟机和 Kubernetes 上的容器等。你可以在所有主要的云供应商、云平台、企业内部和混合云基础设施上运行 Presto。

11.6 其他集成案例

目前，你只看到了一小部分能够使用或集成 Presto 的工具和平台，但仅是能与 Presto 一起使用的商业智能和报告工具就非常多了，至少包括以下这些：

- Apache Superset
- DBeaver
- HeidiSQL
- Hue
- Information Builders
- Jupyter Notebook
- Looker

- MicroStrategy
- Microsoft Power BI
- Mode
- Redash
- SAP Business Objects
- SQuirreL SQL Client
- Tableau
- Toad

使用或支持 Presto 的数据平台、托管平台和其他系统包括：

- AWS 和 Amazon Elastic Kubernetes Service
- Amazon EMR
- Google Kubernetes Engine
- Microsoft Azure Kubernetes Service
- Microsoft Azure HDInsight
- Qlik
- Qubole
- Red Hat OpenShift

其中许多用户或供应商（比如 Qubole）在项目中做出了贡献。

11.7 自定义集成

Presto 是一个开放的平台，你可以在它上面整合自己的工具。围绕 Presto 的开源社区正在积极构建和改进这些集成。

你可以使用 Presto CLI 或 Presto JDBC 驱动进行简单的集成，更高级的集成是使用 Presto 协调器暴露的基于 HTTP 的协议来执行查询。JDBC 驱动只是简单地封装了这个协议，你可以在 Presto 网站上找到包括 R 和 Python 等其他平台的封装。

还有许多组织更进一步，它们为 Presto 实现了新的插件。这些插件可以增加一些功能，如连接到其他数据源的连接器、事件监听器、访问控制、自定义类型和用户自定义函数等，可用于查询语句。Presto 文档中包含了一个非常有用的开发者指南，它可以作为你的首选参考资源。最后别忘了向社区寻求帮助以及给予反馈，参见 1.4.3 节。

11.8 小结

Presto 的应用是如此广泛，你可以将许多工具与 Presto 集成来创建一些非常强大的解决方案，这着实令人惊叹。在本章中，我们仅仅了解了冰山一角。

多亏了 JDBC 驱动、ODBC 驱动、Presto CLI 以及建立在这些工具和其他扩展之上的集成，许多工具可以与 Presto 一起使用。

无论你喜欢使用哪种商业或开源的商业智能报告工具或数据分析平台，请务必确认它们是否支持或可以集成 Presto。同样，你也不妨去了解一下 Presto 是否用于你的工具链底层中，这可以让你更好地了解如何改进或扩展你的使用方式，甚至让你去思考是否可以直接使用 Presto。

根据 Presto 部署的所有权级别，你可以根据需要进行定制、更新和扩展，或者作为集成的一部分，你也可以让供应商管理 Presto。找到属于你自己的理想选择，然后尽情享受 Presto 吧。如果你打算自己部署管理 Presto，可以在第 12 章中了解到更多信息。

第12章

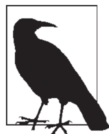
生产环境中的Presto

在学习 Presto 的过程中，第 2 章介绍了探索性安装 Presto 的简单方法，第 5 章介绍了 Presto 的部署，而这一章将更进一步地深入细节。毕竟，仅仅安装和配置集群与维持其长时间运行非常不同，后者具有不同的用户、不断变化的数据源和完全独立的用法。

因此，这一章将进一步探索 Presto 集群操作者必备的其他方面的知识。

12.1 使用Presto Web UI监控

如 3.5 节所述，你可以在每个 Presto 集群的协调器上访问到其 Web UI，它可以用来检查和监控 Presto 集群及其处理的查询。Web UI 所提供的细节信息可用于帮你更好地理解和调优 Presto 的整体系统以及单个查询。



Presto 的 Web UI 暴露的信息来自 Presto 的系统表，在 8.2 节中有详细介绍。

当你第一次访问 Presto Web UI 所在地址时，可以看到如图 12-1 所示的主仪表盘，其上方显示 Presto 集群的信息，下方显示查询列表。

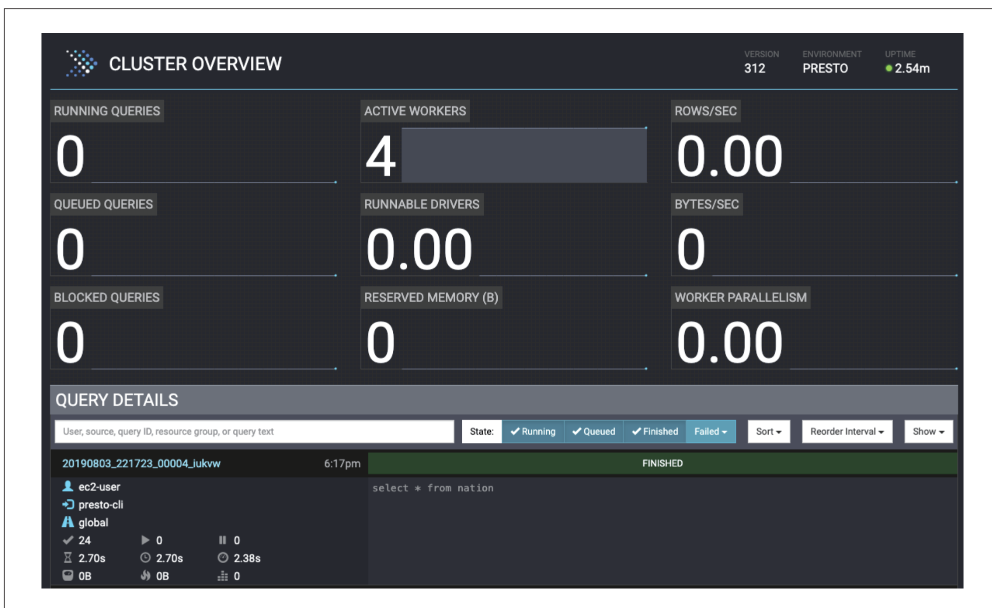


图 12-1：Presto Web UI 的主仪表盘

12.1.1 集群级的细节

下面先来讨论 Presto 集群的信息。

运行中的查询

Presto 集群中当前正在运行的查询总数。所有用户的查询都会计入，例如，Alice 运行着 2 个查询且 Bob 运行着 5 个查询，则在对应方框里显示的总数是 7。

队列中的查询

Presto 集群中所有用户在队列中等待的查询总数。基于资源组的配置，协调器调度在队列中等待的查询。

阻塞的查询

集群中阻塞的查询总数。由于缺少必要的切片或其他资源，阻塞的查询无法继续执行。在接下来的章节里，你会了解更多有关查询状态的知识。

活动的工作节点

集群中活动的工作节点的数量。通过手动或自动扩展添加或删除任何工作节点时，这些节点都会注册到发现服务上，因此显示的数字会据此更新。

可运行的驱动

集群中可运行的驱动的平均数，如第 4 章所述。

保留内存

Presto 中所有保留内存的量，以字节为单位。

行每秒

集群上运行着的所有查询每秒处理的总行数。

字节每秒

集群上运行着的所有查询每秒处理的总数据量，以字节为单位。

工作节点并行度

工作节点的并行度总量，也就是集群中所有查询运行时在所有工作节点上花费的线程 CPU 时间总量。

12.1.2　查询列表

Presto Web UI 仪表盘的下方是最近执行的查询列表，示例截图如图 12-2 所示。在历史列表中展示的查询数量由 Presto 集群的配置决定。

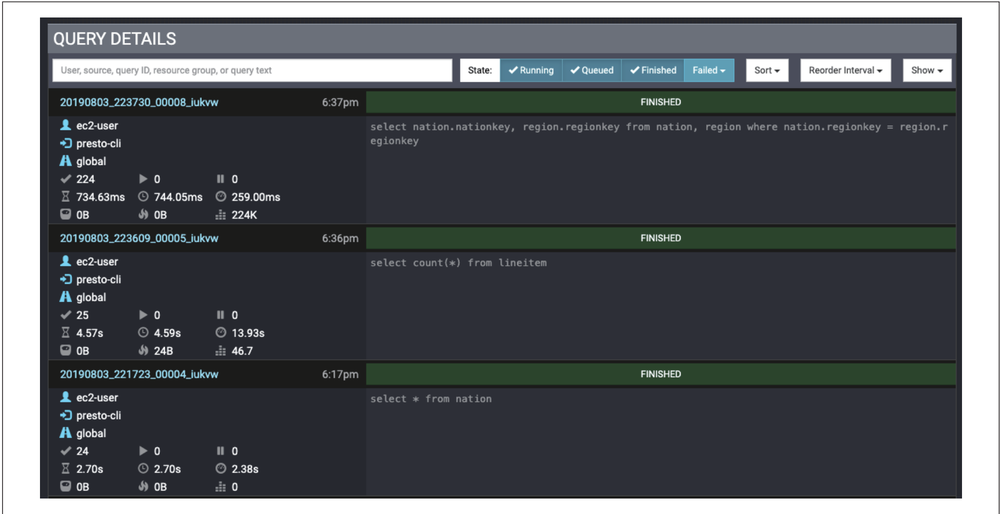


图 12-2：Presto Web UI 中的查询列表

在列表上方，有一系列控件可用于设置列出查询的过滤条件。这可以让你在集群非常繁忙，运行着几十上百条查询时，也能定位到某一特定的查询。

文本输入框可以让你通过键入要使用的标准文本查找特定的查询。可以搜索的内容包括查询发起者的用户名、查询数据源、查询 ID 和资源组等，甚至还包括查询的 SQL 语句文本和查询状态。文本输入框旁边的**状态过滤器**允许你根据查询的状态（执行中、队列中、已完成和失败等）过滤查询列表。对于失败的查询，还可以使用失败原因（内部错误、外部

错误、资源错误或用户错误）进行过滤。

左边的控件允许你设定查询显示的顺序、当数据改变时重排序的时机和显示查询的最大数量。

在查询管理控件下方的每一行都代表一个查询。左边的列显示查询的信息，右边的列显示查询的 SQL 文本和执行状态。查询信息概要的例子如图 12-3 所示。

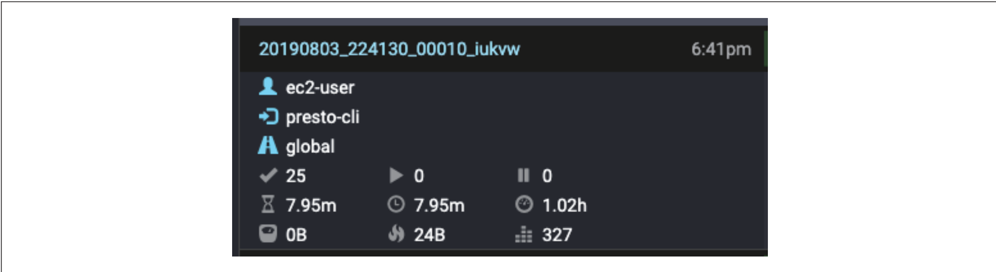


图 12-3：Presto Web UI 中某一查询的信息

让我们更仔细地查看查询信息的细节，每个查询执行时都拥有同样的查询信息。在左上方的文字就是查询 ID，在本例中，查询 ID 的值是 `20190803_224130_00010_iukvw`。更进一步，你可以发现 YYYYMMDD_HHMMSS 格式的日期和时间（协调世界时，UTC）组成了 ID 的前半部分；ID 的后半部分是查询的一个递增的计数器，计数器值 `00010` 仅仅代表着，这是自协调器启动以来第 10 个运行的查询；ID 的最后一部分（`iukvw`）是协调器的随机标识符。随机标识符和计数器值在协调器重启时都会重置。右上方的显示的时间是查询运行时的本地时间。

例子中接下来的三个值（`ec2-user`、`presto-cli` 和 `global`）代表着运行此查询的终端用户、查询来源和用户组。本例中，用户名是默认的 `ec2-user`，查询来源是 `presto-cli`。如果你在运行 Presto CLI 时指定了 `--user` 参数，这里显示的将是你指定的用户名。查询来源也可以不是 `presto-cli`，例如，当你使用 JDBC 驱动连接到 Presto 时，它会显示 `presto-jdbc`。使用 Presto CLI 的 `--source` 参数或 JDBC 连接的字符串属性，你也可以将查询来源设定成任何你想要的值。

下方值的阵列在 Presto Web UI 中并未详细标注，但是它们包含了有关查询的一些重要信息。表 12-1 解释了这些值的含义。

表12-1：用于特定查询的值阵列

已完成切片数：查询已完成的切片数。例子中显示 25 个切片已经执行完成。在查询开始时，这个值是 0，随着查询的执行和切片的完成，该值会不断上升	**运行中切片数**：运行中的查询切片数。查询完成后，该值是 0，但在查询执行时，该值随着切片开始运行和完成而动态变化	**队列中切片数**：查询执行时在队列中的切片数量。查询完成后这个值是 0，但在查询执行时，这个值随着切片在队列和执行状态之间切换而动态变化

（续）

挂钟时间：执行查询已花费的总时间。这个值在你分页查看结果时也会增长	总挂钟时间：这个值与挂钟时间类似，但包含了队列时间。挂钟时间本身不包含查询在队列中等待的时间。总挂钟时间是你从提交查询到接收完结果所观察到的总时间	CPU 时间：处理查询所花费的总 CPU 时间。这个值通常比挂钟时间大，因为并行的工作节点及其线程的时间都独立统计并累加起来。例如，4 个 CPU 花费 1 秒执行 1 个查询，则总 CPU 时间是 4 秒
当前总保留内存：当前查询执行所使用的总保留内存的量。对于已完成的查询，这个值为 0	峰值总内存：在查询执行过程中所使用的峰值总内存量。查询执行中的特定操作可能需要很多内存，因此了解峰值内存非常有用	累计用户内存：在整个查询执行过程中使用的累计用户内存量。这不意味着这些内存在某一时刻同时被使用。这个值是内存的累计使用量



Presto Web UI 上的许多图标和值在鼠标悬停时会显示弹出工具提示。当你不确定某一特定值的含义时，这一功能十分有用。

接下来你需要学习的是查询处理的不同状态，这些状态显示在查询文本上方。最常见的查询状态包括 `RUNNING`（执行中）、`FINISHED`（已完成）、`USER CANCELLED`（用户取消）或 `USER ERROR`（用户错误）等。`RUNNING` 和 `FINISHED` 状态的含义无须多言；`USER CANCELLED` 表示用户终止了查询；而 `USER ERROR` 表示用户提交的 SQL 查询包含语法错误或语义错误。

当查询在执行过程中因等待要处理的资源或额外的切片等原因阻塞时，查询会进入 `BLOCKED`（阻塞）状态。观察到一个查询进入或离开此状态十分正常。然而，如果一个查询一直卡在阻塞状态，可能的原因有很多，并且可能表明查询或 Presto 集群内部存在问题。如果你发现查询卡在阻塞状态，请先检查系统的内存使用情况和配置。原因可能是，此查询需要的内存非同寻常地高，或者在计算上代价非常大。此外，如果客户端没有获取结果或不能以足够的速度读取结果，反向压力也会将查询置于 `BLOCKED` 状态。

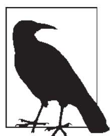
当查询开始或停止处理，并由资源组定义的规则置于等待阶段时，`QUEUED`（队列中）状态就会发生。查询此时仅仅是在等待执行。

你也可能见到 `PLANNING`（规划中）状态下的查询。这通常发生在需要很多优化工作的大而复杂的查询上。如果你经常看到此状态，且查询优化似乎占据了可观的查询执行时间，你应该调查可能的原因，如协调器内存不足或处理能力不足等。

12.1.3 查询细节视图

目前，你已经看到了有关 Presto 集群的总体信息和查询的概述信息。Web UI 还提供了针对每个查询更深入的细节信息。在图 12-3 中所示的界面上单击特定查询的名称即可查看查询细节（Query Details）视图。

查询细节视图包含大量关于该查询的信息，让我们仔细探索其中的内容。



查询细节视图通常提供给 Presto 开发者和深度用户使用，这个层级的复杂度需要你非常熟悉 Presto 代码及其内部构造。检查这个视图对普通用户仍是有用的，随着时间的推移，你可以从中学到很多专业知识。

查询细节视图使用多个标签页展示 Presto 查询不同方面的细节信息。在标签页之外，查询 ID 和状态一直可见，图 12-4 展示了此视图头部的示例。

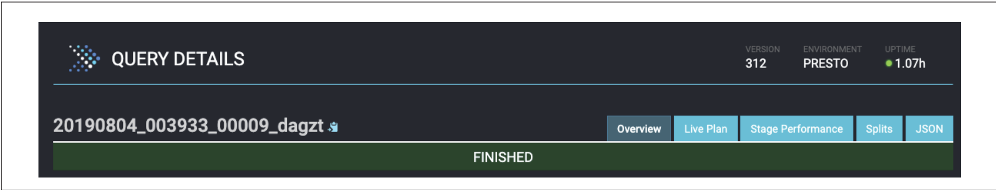


图 12-4：查询细节视图的头部和标签页部分

1. 概览

概览页包含了查询列表中查询细节区域显示的信息，并提供了更多其他方面的信息：

- 会话（Session）
- 执行（Execution）
- 资源利用概览（Resource Utilizations Summary）
- 时间线（Timeline）
- 查询（Query）
- 准备查询（Prepare Query）
- Stages
- 任务（Tasks）

如图 12-5 所示，在 Stages 区域展示了查询 Stage 的信息。

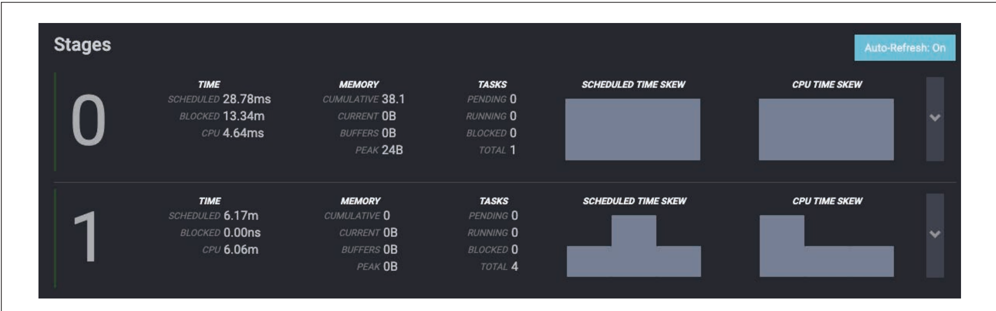


图 12-5：查询细节页面概览标签页中的 Stages 区域

上述例子展示的查询语句是 `SELECT count(*) FROM lineitem`。这是一个简单的查询，它只包括两个 Stage：Stage 0 是一个单任务的 Stage，它运行在协调器上，负责将 Stage 1 中的结果合并起来，并完成最终的聚合；Stage 1 是一个将任务运行在各个工作节点上的分布式 Stage，负责读取数据并计算局部聚合。

下述列表解释了 Stages 区域中每个 Stage 上显示的数值的含义。

TIME—SCHEDULED
: 在此 Stage 的所有任务完成前，它处于调度中状态的总时间。

TIME—BLOCKED
: 此 Stage 因等待数据而被阻塞的总时间。

TIME—CPU
: 此 Stage 中任务花费的总 CPU 时间。

MEMORY—CUMULATIVE
: 运行此 Stage 所花费的累计内存总量。这个值并不意味着某一时刻 Stage 使用了这么多的内存，它表示的是在处理过程中使用到的内存的累加量。

MEMORY—CURRENT
: 此 Stage 当前使用的总保留内存量。对于已完成的查询，这个值显示为 0。

MEMORY—BUFFERS
: 当前等待处理的数据占用的内存量。

MEMORY—PEAK
: 执行此 Stage 使用到的峰值总内存量。查询执行过程中的某些操作可能需要很多内存，因此了解峰值内存使用量非常有用。

TASKS—PENDING
: 此 Stage 中处于等待状态的任务数量。查询结束后，这个值总是显示为 0。

TASKS—RUNNING
: 此 Stage 正在执行的任务数量。查询结束后，这个值总是显示为 0。在查询执行时，这个值根据任务开始和完成的情况动态更新。

TASKS—BLOCKED
: 此 Stage 中处于阻塞状态的任务数量。查询结束后，这个值总是显示为 0。在查询执行时，这个值根据任务在阻塞和执行状态之间的切换情况动态更新。

TASKS—TOTAL
: 查询完成的总任务数量。

SCHEDULED TIME SKEW、CPU TIME SKEW、TASK SCHEDULED TIME 和 TASK CPU TIME

这些直方图显示了调度时间、CPU 时间、任务调度时间和多工作节点多任务的 CPU 时间等指标的分布和变化趋势，这可以让你在执行长时间、分布式查询时诊断工作节点的利用情况。

如图 12-6 所示，Stages 区域下方的区域描述了更多任务细节。

Tasks

Show ▾ Auto-Refresh: On

ID	Host	State					Rows	Rows/s	Bytes	Bytes/s	Elapsed	CPU Time	Buffered
0.0	172.31.47.164	FINISHED	0	0	0	17	8.00	0.17	248	5.26	47.12s	4.64ms	0
1.0	172.31.36.102	FINISHED	0	0	0	2	150M	3.30M	0	0	45.42s	1.49m	0
1.1	172.31.37.8	FINISHED	0	0	0	2	150M	3.18M	0	0	47.13s	1.54m	0
1.2	172.31.32.176	FINISHED	0	0	0	2	150M	3.23M	0	0	46.42s	1.52m	0
1.3	172.31.47.164	FINISHED	0	0	0	2	150M	3.23M	0	0	46.37s	1.51m	0

图 12-6：查询细节页面上的任务信息区域

表 12-2 给出了上述任务列表中各个值的含义。

表12-2：图12-6任务列表中列的描述

列	描　　述
ID	任务标识符，格式为 `stage-id.task-id`。例如，ID 0.0 表示 Stage 0 的任务 0，1.2 表示 Stage 1 的任务 2
Host	运行此任务的工作节点的 IP 地址
State	任务的状态，可能的值有 PENDING、RUNNING 或 BLOCKED
Pending Splits	任务处于等待（PENDING）状态的切片数量。这个值在任务执行时不断变化，直到任务结束时变为 0
Running Splits	正在运行（RUNNING）的任务切片数量。这个值在任务执行时不断变化，直到任务结束时变为 0
Blocked Splits	阻塞（BLOCKED）状态的切片数量。这个值在任务执行时不断变化，直到任务结束时变为 0
Completed Splits	已完成的任务切片数量。这个值在任务执行时不断变化，直到任务结束时等于总的切片数量
Rows	任务处理过的总行数，该值在任务执行时不断上升
Rows/s	任务每秒处理的行数
Bytes	任务处理的总字节数，该值在任务执行时不断上升
Bytes/s	任务每秒执行的字节数
Elapsed	任务经过的挂钟时间总长度
CPU Time	任务花费的总 CPU 时间
Buffered	当前在缓冲区中等待处理的数据量

如果你仔细查看这些值，就可以发现它们是如何累计的。例如，所有任务的总 CPU 时间加起来，等于它们所属 Stage 的总 CPU 时间。所有 Stage 的总 CPU 时间加起来，等于查询花费的总 CPU 时间。

2. 实时计划

实时计划（Live Plan）标签页可以让你实时观察 Presto 集群执行查询处理的过程，示例如图 12-7 所示。

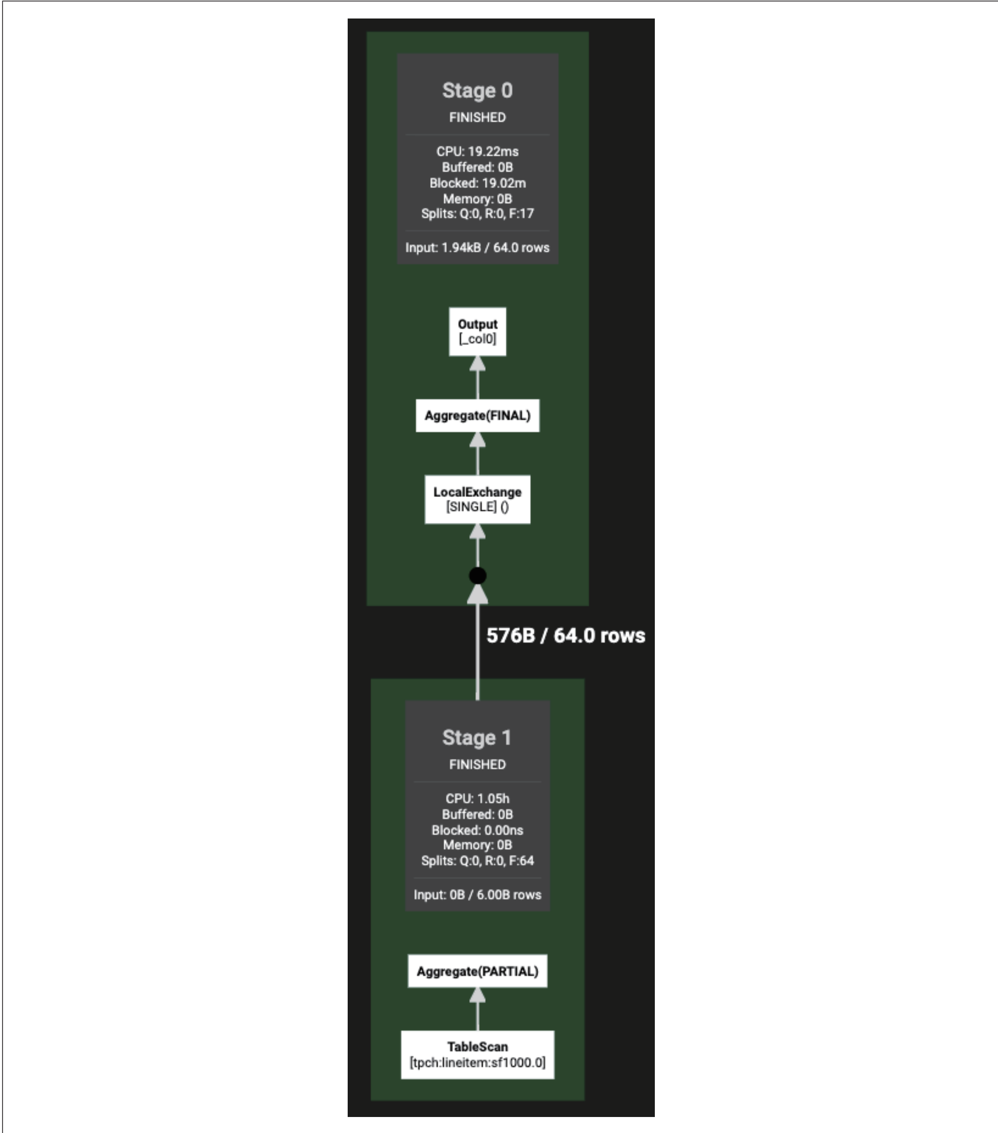


图 12-7：在 `lineitem` 上执行 `count(*)` 查询的实时计划示例

在查询执行时，图 12-7 中所示查询计划上的计数器值会随着查询的执行而更新。执行计划上显示的值的含义与概览（Overview）标签页中所描述的相同，区别在于它们被放置在查询计划图上并实时更新。当查询卡住或消耗时间过长时，该视图能以可视化的方式帮助你诊断、解决性能问题。

3. Stage 性能

在查询处理结束后，Stage 性能视图可以提供有关查询性能细节的可视化，示例如图 12-8 所示。

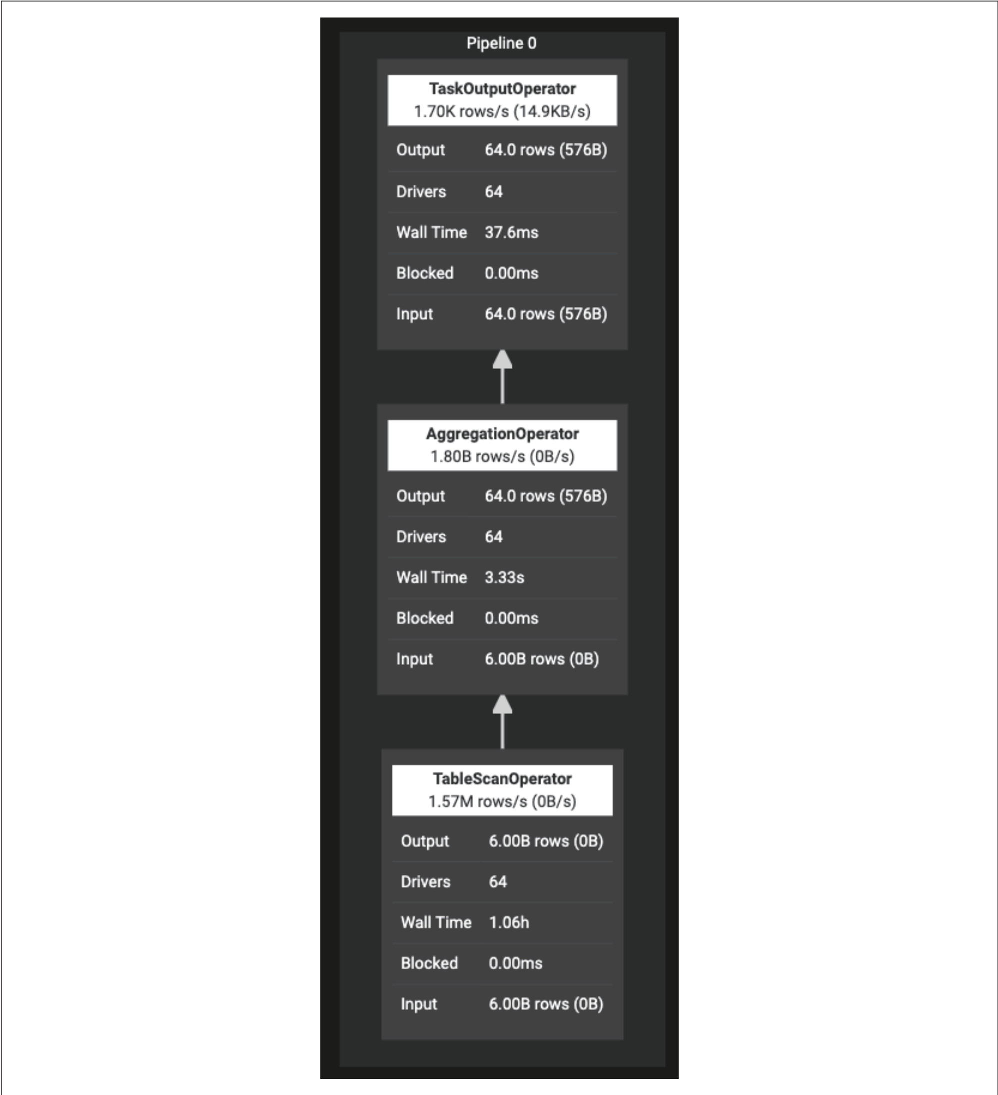


图 12-8：`count(*) lineitem` 查询的 Stage 性能视图

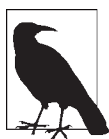
这个视图可以看作实时计划视图的进一步下钻，在这里你可以看到 Stage 内任务的算子管线。计划图上显示的值与概览（Overview）标签页上描述的相同。从这个视图也可以发现查询在哪里卡住或者哪里是性能瓶颈，以便于诊断或解决性能问题。你可以单击独立的算子来获得更详细的信息。

切片

切片视图显示了在查询执行过程中切片创建和执行的时间线。

JSON

JSON 标签页以 JSON 格式提供了所有查询的细节信息，这些信息根据获取快照的不同而改变。



Web UI 上的一部分组件为复制长文本到系统剪切板提供了方便。请留意界面上的剪切板图标，当你单击它时，可以复制相关联的文本到系统剪切板，以便你在其他地方粘贴并使用。

12.2 Presto SQL查询调优

在 4.8 节，你已经了解了 Presto 的基于代价的优化器。回顾一下，SQL 是一门声明式的语言，用户编写 SQL 查询来描述他们想要的数据。和命令式程序不同，在 SQL 中，用户并不指定如何处理数据并获得结果，获得所需结果的步骤将交由查询优化器来决定。这里的执行步骤就是所谓的**查询调优**。

大多数情况下，提交 SQL 的终端用户可以依赖 Presto 来计划、优化和执行 SQL 查询，并快速高效地获得结果。终端用户无须关心太多细节。

然而，有时你无法获得期望的性能，因此需要进行 Presto 查询调优。你需要鉴别是某个查询本身存在性能问题，还是具有某些相似属性的多个查询有性能问题。

下面先从单个查询的调优开始，假设系统上其他的查询运行正常。在检查慢查询时，你应该首先检查查询使用到的表是否有数据统计信息。在本文编写时，只有使用 Hive 连接器的表提供了数据统计信息，不过预计其他连接器也会很快开始提供这些信息：

```
presto:ontime> SHOW STATS FOR flights;
```

SQL 中 Join 是成本最高的操作之一，在进行查询性能调优时，你应该重点关注 Join。你可以在查询上使用 `EXPLAIN` 命令来确定 Join 顺序：

```
presto:ontime> EXPLAIN
               SELECT f.uniquecarrier, c.description, count(*) AS ct
               FROM postgresql.airline.carrier c,
                    hive.ontime.flights_orc f
               WHERE c.code = f.uniquecarrier
```

```
GROUP BY f.uniquecarrier, c.description
ORDER BY count(*) DESC
LIMIT 10;
```

通常来说，你希望数据量较小的输入表作为 Join 的构建侧，散列 Join 中这一侧会用来构建散列表。由于构建侧需要读入全部数据并在内存中构建散列表，因此你希望它的输入越小越好。要决定 Presto 是否生成了正确的 Join 顺序，需要进一步研究数据相关的领域知识。如果对数据完全不了解，你可能需要运行一些试验性的查询来获得一些额外信息。如果你认为 Presto 给出的 Join 顺序不是最优的，你可以通过设置一个切换，来使用表在 SQL 中出现的语法顺序来强制指定 Join 顺序。这可以通过在 config.properties 文件中使用 `optimizer.join-reordering-strategy` 属性进行全局设置，但如果你只想指定某一查询的 Join 顺序，则可以使用会话属性 `join_reordering_strategy`（见 8.8 节）。此属性的可选值有 `AUTOMATIC`、`ELIMINATE_CROSS_JOINS` 和 `NONE`，而使用后两个选项值会覆盖 CBO 选择的 Join 顺序。`ELIMINATE_CROSS_JOINS` 是一个不错的折中方案，因为它只会在消除 Cross Join 的时候重排序 Join（通常是最佳实践），而其他表则保留查询编写者指定的顺序：

```
...
FROM postgresql.airline.carrier c,
hive.ontime.flights_orc f
...

...
FROM hive.ontime.flights_orc f,
postgresql.airline.carrier c
...
```

CBO 调优

默认情况下，Presto CBO 一次最多可以重排序 9 个表。对于超过 9 个表的情况，CPO 会将搜索空间分段。如果查询中有 20 个表，则 Presto 会先将前 9 个表作为一个优化问题进行排序，再将跟着的 9 个表作为第二个优化问题进行排序，最后再处理剩下的 2 个表。之所以设置这个限制，是因为可能的 Join 顺序的数量是表数量的阶乘。因此，必须设定一个合理的限制以防搜索空间过大。你可以 config.properties 文件中设定 `optimizer.max-reordered-joins` 参数来提高这个值。提高这个值可能会导致 Presto 花费大量时间和资源进行查询优化，从而导致性能问题。回顾一下，CBO 的目标不是找到最佳的计划，而是足够好的计划。

除了 Join 优化之外，Presto 还包含一些启发式的优化。这些优化不是基于代价的，因此不一定总能获得最佳的结果。优化器可以利用 Presto 是一个分布式查询引擎这一事实，使聚合可以并行运行。这意味着聚合可以被切分成多个更小的部分，从而分发到多个工作节点上并行运行，并在最终步骤从局部结果聚合起来。

Presto 和其他 SQL 引擎常用的一个优化是，将部分聚合下推到 Join 之前执行，从而减少输入 Join 的数据量。要使用这个优化，可以设置 `push_aggregation_through_join` 属性。由于下推的聚合只产生局部结果，因此，在 Join 之后仍需要执行一次最终聚合。使用这个优化可以获得的收益依赖于实际的数据，有时它甚至会产生更慢的查询。例如，当 Join 条件比较严格，只会产生少量数据时，在 Join 之后再进行聚合往往更高效。可以在当前会话上将这个属性值设为 `false`，以关闭此优化并进行性能试验。

另一种常见的启发式优化是在计算最终聚合之前先计算局部聚合：

```
presto:ontime> EXPLAIN SELECT count(*) FROM flights_orc;
                             Query Plan
---------------------------------------------------------------------
 - Output[_col0]
         Layout: [count:bigint]
         _col0 := count
     - Aggregate(FINAL)
             Layout: [count:bigint]
             count := "count"("count_3")
         - LocalExchange[SINGLE] ()
                 Layout: [count_3:bigint]
             - RemoteExchange[GATHER]
                     Layout: [count_3:bigint]
                 - Aggregate(PARTIAL)
                         Layout: [count_3:bigint]
                         count_3 := "count"(*)
                     - TableScan[hive:ontime:flights_orc]
                             Layout: []
(1 row)
```

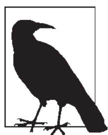
尽管这通常是一种很好的启发式算法，用于存放散列表所用的内存的量仍可以进一步调优。如果表中有很多行，分组键（`GROUP BY`）中不同的值很少时，这个优化会显著减少在网络上传输的数据量，因此非常有效；但如果不同的值很多，就需要更大的散列表来存放这些数据。默认情况下，散列表使用 16MB 的内存，但你可以在 config.properties 文件中设定 `task.max-partial-aggregation-memory` 属性来调整这个值。但是，如果不同的分组键数量太多，聚合并不能减少网络传输的数据量，甚至可能拖慢查询速度。

12.3 内存管理

想要正确地配置和管理 Presto 集群的内存并不容易。许多持续变化的因素会影响内存需求：

- 工作节点的数量；
- 协调器和工作节点的内存；
- 数据源的数量和类型；
- 所执行的查询的特性；
- 用户的数量。

对于 Presto 这样的多用户、多工作节点系统，资源管理是一个极具挑战性的任务。基本上来说，你要先选择一个起始点，再通过不断地监控并调整系统，使其满足现在和未来的需求。让我们深入细节，讨论一些关于在 Presto 中管理和监控内存的建议和指导方针。



这里讨论的所有内存管理方法都应用于运行 Presto 工作节点的 JVM。这些内存都是在工作节点的 JVM 中分配的，因此 JVM 本身的配置也需要考虑到这些值，以容纳它们所设定内存分配量。

根据并行运行的查询数量，JVM 可用的内存可能需要设定为一个更大的值，下面的例子说明了这一点。

之前提到的所有影响因素都可以合并为**工作负载**（workload）这一因素进行考量。集群内存的调优重度依赖上面运行的工作负载。

例如，大多数查询包含多个 Join、聚合和窗函数。如果查询的工作负载轻，你就可以设定降低的单个查询内存限制并提高并发性；反过来，如果查询工作负载重，你就只能提高单个查询的内存限制并降低并发性。近似来讲，查询的规模相当于查询本身的特性和输入数据量的乘积。Presto 提供了一系列通过 config.properties 文件设定的属性值，可以让你在部署时管理集群的内存使用：

- `query.max-memory-per-node`
- `query.max-total-memory-per-node`
- `query.max-memory`
- `query.max-total-memory`

Presto 管理的内存被分为以下两种类型。

用户内存

用户查询中的聚合和排序等操作会影响用户内存的分配。

系统内存

系统内存的分配基于查询引擎自己的执行实现。缓冲区的读写和 shuffle、表扫描等操作会影响系统内存的分配。

了解了这一分类，你就可以进一步理解以下内存属性了。

`query.max-memory-per-node`

一个查询在单个工作节点上可以使用的**用户**内存的最大值。这些内存可用于处理聚合和输入数据分配等。

`query.max-total-memory-per-node`

一个查询在单个节点上可用的内存总量（包括用户内存和系统内存）的最大值，这个值必须大于 `query.max-memory-per-node`。当一个查询消耗的用户内存和系统内存总量超

过了这个限制，它就会被终止。

`query.max-memory`

一个查询在全集群所有工作节点上可以使用的**用户**内存总量的最大值。

`query.max-total-memory`

一个查询可以在全集群所有节点上使用的内存总量（包括用户内存和系统内存）的最大值，因此这个值必须大于 `query.max-memory`。

如果一个查询因超过限制而被终止，系统将返回下列错误代码以说明原因。

- `EXCEEDED_LOCAL_MEMORY_LIMIT` 表示内存使用超过了 `query.max-memory-per-node` 或 `query.max-total-memory-per-node` 的限制。
- `EXCEEDED_GLOBAL_MEMORY_LIMIT` 表示内存使用超过了 `query.max-memory` 或 `query.max-total-memory` 的限制。

我们来看一个真实的例子，有个小集群包含 1 个协调器和 10 个工作节点，它的特性如下所示。

- 1 个协调器。
- 10 个工作节点，通常每个工作节点都具有相同的系统规格。
- 每个工作节点拥有的物理内存：50GB。
- `jvm.config` 在 `-Xmx` 中设置的 JVM 堆内存上限是 38GB。
- `query.max-memory-per-node`：13GB。
- `query.max-total-memory-per-node`：16GB。
- `memory.heap-headroom-per-node`：9GB。
- `query.max-memory`：50GB。
- `query.max-total-memory`：60GB。

下面进一步分解这些数字。

每个工作节点上的可用内存总量约 50GB，我们为操作系统、代理 / 后台进程以及系统上 JVM 外的组件保留了约 12GB 的内存，监控系统和其他允许你管理机器并使其正常运行的程序会使用这 12GB 内存。因此，我们将 JVM 堆内存上限设为 38GB。

当查询规模较小时，并发性可以设置得大一些。在前面的例子中，我们假设查询从中等规模到大规模不等，并且可能存在数据倾斜。一个查询在集群整体可用的用户内存 `query.max-memory` 被设为 50GB。在讨论 `max-memory` 的同时，我们还需要考虑**原始散列分区数**（initial-hash-partitions），这个值在理想情况下应该小于等于工作节点的数量。

如果我们将原始散列分区数设为 8，`max-memory` 设为 50GB，则每个节点得到的用户内存配额就大约为 50GB/8 = 6.25GB。我们将节点本地的限制 `max-memory-per-node` 设为 13GB，

在允许有数据倾斜的情况下，一个节点最多可以消耗平均内存的两倍。内存消耗量这样的数字根据数据组织的方式和常见查询的类型（也就是集群的工作负载）的不同，会产生显著的变化。此外，集群的基础设施，如机器的大小和数量，也对配置选择有巨大的影响。

为了防止死锁，你可以设置 `query.low-memory-killer.policy` 属性。可选的值是 `total-reservation` 或 `total-reservation-on-blocked-nodes`。选用 `total-reservation` 时，Presto 会终结集群中占用内存最多的查询，从而释放资源。另外，选用 `total-reservation-on-blocked-nodes` 时，系统会终止在阻塞的节点上使用最多内存的查询。

从例子中可以看出，你可以先根据一些假设条件开始设定参数，然后再根据实际的工作负载进行调整。

例如，为可视化工具上的交互式 ad-hoc 查询所准备的集群，可能需要执行大量小规模的查询。随着用户数量的增多，查询数量和并发性都会上升。通常情况下，并不需要更改内存配置，只需要为集群添加工作节点即可。

然而，当同一个集群接入一个拥有海量数据的数据源并在上面执行复杂查询时，这些查询就很可能会超出资源的使用限制。此时，你就必须要调整内存配置了。

这将引入另一个值得一提的点。作为最佳实践，通常 Presto 集群应该具有完全相同的工作节点，而所有的工作节点都应具有相同的虚拟机（VM）镜像、容器大小，以及相同的硬件规格。因此，更改工作节点上的内存通常意味着，新值对于物理内存来说太大或太小，无法很好地利用整个系统。因此，调整内存就需要替换集群中的所有工作节点。基于你使用的集群基础设施，如私有云中的虚拟机或公有云中的 Kubernetes 集群容器，进行这类操作的难易程度也不一样。

这也引出了最后一个值得讨论的观点。你的工作负载可能差异巨大：有很多是小的、快速且占用很少内存的 ad-hoc 查询，另一些则是庞大的、长时间运行、包含各种分析，甚至使用不同数据源的查询。不同的工作负载需要完全不同的内存设置，甚至不同的工作节点设置和监控需求。在这种场景下，你应该采取进一步的操作，通过使用不同的 Presto 集群来分离工作负载。

12.4 任务并发性

为了提高 Presto 集群的性能，你需要修改一些任务相关的属性的默认值。本节我们讨论需要调整的两个常见属性，不过你还可以从 Presto 文档中找到一些其他可以调整的属性，这些属性都在 config.properties 中设定。

任务工作线程数

默认值是机器 CPU 数的 2 倍。例如，一台具有 2 个 6 核 CPU 的机器使用 2 × 6 × 2 = 24 个工作线程。如果你观察到所有的线程都在使用，但 CPU 使用率仍很低，则可以尝试

借助 task.max-worker-threads 属性来增加线程数，以此提高 CPU 利用率和性能。建议尝试慢慢增加这个值，因为将其设置得过高会带来更高的内存使用和额外的上下文切换成本，这反而会带来负面效果。

任务的算子并发数

Join 和 Aggregation 之类的算子通过本地数据分区和并行执行算子来实现并行处理。例如，数据先在本地根据 GROUP BY 的列进行分区，之后多个 Aggregation 算子会并行地进行处理。这些并行操作的默认并发数是 16，你可以通过设定 task.concurrency 属性来调整它。如果你运行很多并发的查询，上下文切换的成本可能会导致性能下降。对于只运行少量并发查询的集群，更高的并发数可以提高并行度并因此提高性能。

12.5 工作节点调度

你可以通过调整某些调度相关属性的默认值来提高 Presto 集群的性能。通常有 3 个常见的配置可以调整：

- 每个任务的切片数；
- 每个节点的切片数；
- 本地调度策略。

Presto 文档还介绍了许多其他相关属性。

12.5.1 根据任务或节点调度切片

每个工作节点能处理的切片数量有上限，默认情况下其最大值是 100。你可以通过 node-scheduler.max-splits-per-node 属性调整这个值。当发现工作节点上的切片数已经达到此限制但仍没有充分利用资源时，你可以考虑调整这个值。提高该值通常会提高性能，特别是存在很多切片的情况下。此外，你还可以提高 node-scheduler.max-pending-splits-per-task 属性的值，这个属性的值不应该超过 node-scheduler.max-splits-per-node 属性的值。它确保在工作节点处理任务的同时，等待的任务可以被放入队列中。

12.5.2 本地调度策略

Presto 调度器使用两种方法调度工作节点上的切片，具体使用哪种方法依赖于 Presto 的部署情况。如果工作节点和分布式存储节点部署在相同的机器上，则最好将切片调度在数据所在的工作节点上；如果将切片调度在其他节点而不是数据所在节点，则这些数据必须在网络上传输才能处理。因此，将切片调度到数据存储的节点上可以提高性能。

你可以使用 node-scheduler.network-topology 属性设定切片调度策略，此属性的默认值 legacy 在调度切片时不会考虑数据位置。你可以将其修改为 flat，从而使用考虑数据位置

的改进策略。Presto 调度器会使用队列 50% 的空间来调度本地切片。

当 Presto 安装并放置在 HDFS 数据节点上，或使用 RubiX 缓存（参见 11.2 节）时，适合使用 flat 策略。RubiX 将分布式存储中的数据缓存在工作节点上，因此，调度本地切片会提供更好的性能。

12.6 网络数据交换

网络配置和设置以及与数据源的距离也是影响 Presto 集群性能的重要因素。Presto 支持一些网络特定的属性，使你可以修改默认值并匹配特定的使用场景。

除了提高性能以外，你有时还需要调优一些其他的网络相关问题，以使你的查询运行良好。下面将讨论可以调优的一些常见的属性。

12.6.1 并发性

Presto 的数据交换客户端负责请求上游任务产生的数据，它使用的默认线程数是 25。你可以通过设定 `exchange.client-threads` 属性来修改这个值。

使用更多的线程可以提高大规模和高并发配置集群的性能。这是因为在这些集群中，产生数据的速度更快，提高用于消费数据的并发性可以降低时延。记住，更多的线程也会需要更多的内存。

12.6.2 缓冲区大小

在数据交换时，发送方和接收方会将收发的数据放在缓冲区中。发送方和接收方的缓冲区大小可以分别配置。

在发送方，任务将产生的数据写入缓冲区并等待下游的数据交换客户端进行请求。默认的缓冲区大小是 32MB，这可以通过 `sink.max-buffer-size` 属性调整。提高这个值可能有助于提高大规模集群的吞吐量。

在接收方，在下游任务处理数据前，数据先放置在缓冲区中。这个缓冲区的默认大小也是 32MB 且可以通过 `exchange.max-buffer-size` 属性进行调整。设定较大的接收缓冲区可以从网络上获取更多数据来延缓**背压**（back pressure）的发生，从而提高查询性能，特别是大规模集群的性能。

12.7 JVM调优

在第 2 章中，你已经知道可以使用 etc/jvm.config 配置文件来配置 JVM 的命令行选项。Presto 启动器使用这个文件的内容启动运行 Presto 的 JVM。相比前面提到的配置，更适合

生产环境下使用的配置会设定更高的内存限制：

```
-server
-XX:+UseG1GC
-XX:+ExplicitGCInvokesConcurrent
-XX:+ExitOnOutOfMemoryError
-XX:+UseGCOverheadLimit
-XX:+HeapDumpOnOutOfMemoryError
-XX:-UseBiasedLocking
-Djdk.attach.allowAttachSelf=true
-Xms16G
-Xmx16G
-XX:G1HeapRegionSize=32M
-XX:ReservedCodeCacheSize=512M
-Djdk.nio.maxCachedBufferSize=2000000
```

你经常需要设定 `Xmx` 属性的值来扩大 JVM 的最大内存分配池（在本例中，这扩大到 16GB）。使用 `Xms` 参数可以设定初始的最小内存分配池大小。通常建议将 `Xmx` 和 `Xms` 设定为相同值。

前面的配置示例将内存设定为 16GB，实际值应该基于你的集群使用的机器的内存容量。通常推荐将 `Xmx` 和 `Xms` 都设置为系统总内存的 80%，这为其他的系统进程预留出很多空间。更多有关 Presto 内存管理的配置参见 12.3 节。

对于大规模的 Presto 部署，分配大于 200GB 的内存并不罕见。

Presto 可以与监视程序这样的小程序运行在相同的机器上，但强烈建议不与其他资源密集的软件共享系统。例如，Presto 和 Apache Spark 不应该运行在同一组硬件上。

如果你怀疑 Java 垃圾回收（GC）有问题，可以添加额外的参数来调试：

```
-XX:+PrintGCApplicationConcurrentTime
-XX:+PrintGCApplicationStoppedTime
-XX:+PrintGCCause
-XX:+PrintGCDateStamps
-XX:+PrintGCTimeStamps
-XX:+PrintGCDetails
-XX:+PrintReferenceGC
-XX:+PrintClassHistogramAfterFullGC
-XX:+PrintClassHistogramBeforeFullGC
-XX:PrintFLSStatistics=2
-XX:+PrintAdaptiveSizePolicy
-XX:+PrintSafepointStatistics
-XX:PrintSafepointStatisticsCount=1
```

这些选项在调试 full GC 停顿问题时非常有用。由于 JVM 本身的进步和 Presto 查询引擎的工程改良，GC 停顿应该很少发生。如果真的发生了，就应该仔细检查原因。修复此类问题的第一步通常是升级 JVM 和 Presto 的版本，因为二者经常更新性能。



JVM 和垃圾回收算法及其配置都是复杂话题。你可以从 Oracle 或其他 JVM 供应商那里获得 GC 调优的文档。我们强烈建议在将这些设置部署到生产环境前，先在测试环境内微调它们。同时请注意，Presto 现在要求使用 Java 11 版本，较新或较老的 JVM 版本以及其他供应商提供的 JVM 都可能表现出不同的行为。

12.8 资源组

资源组（resource group）是 Presto 中用于限制系统资源使用的一个非常有用的概念。资源组的配置包含两个方面：资源组属性和选择器规则。

资源组是一系列定义可用集群资源的命名属性，你可以认为一个资源组是集群中与其他资源组隔离的一部分资源。资源组由 CPU 和内存限制、并发数限制、队列优先级和队列中查询的优先级权重等定义。

另外，选择器规则允许 Presto 将接收到的查询请求分配到特定的资源组。

默认的资源组管理器使用基于文件的配置并通过 etc/resource-groups.properties 文件指定类型和配置文件路径。

```
resource-groups.configuration-manager=file
resource-groups.config-file=etc/resource-groups.json
```

如你所见，真正的配置是一个 JSON 格式的文件。此文件的内容定义了资源组和选择器规则。注意这个 JSON 文件可以放在 Presto 能访问的任何路径上，并且你只需在协调器上配置资源组即可：

```
{
  "rootGroups": [],
  "selectors": [],
  "cpuQuotaPeriod": ""
}
```

`cpuQuotaPeriod` 字段是可选的。

下面先来看两个资源组的定义：

```
  "rootGroups": [
    {
      "name": "ddl",
      "maxQueued": 100,
      "hardConcurrencyLimit": 10,
      "softMemoryLimit": "10%",
    },
    {
      "name": "ad-hoc",
```

```
      "maxQueued": 50,
      "hardConcurrencyLimit": 1,
      "softMemoryLimit": "100%",
          }
  ]
```

上述例子定义了两个资源组，分别名为 `ddl` 和 `ad-hoc`。每个资源组都设定了最大并发查询数和总的分布式内存上限。对于给定的资源组，如果已经达到并发数或内存限制，则新的查询会在队列中等待。一旦总内存使用下降或查询结束，资源组就会从队列中选择一个查询来调度运行。每个资源组的查询队列长度有上限，如果达到了限制，则新的查询会被拒绝并且客户端会收到错误提示。

在我们的例子中，`ad-hoc` 资源组旨在处理非 DDL 查询的所有查询。这个资源组一次只允许一个查询运行，最多可以有 50 个查询在队列中等待。这个资源组的内存限制是 100%，意味着它可以使用集群中的全部内存。

DDL 查询有它们自己的资源组。由于这些类型的查询相对短小和轻量，因此不应该被长时间运行的 ad-hoc SQL 查询阻塞。在这个组中，我们设定了最多有 10 个 DDL 查询并发运行。所有查询使用的分布式内存加起来不应该超过 Presto 集群内存的 10%，这使得 DDL 查询无须与 ad-hoc 查询在同一个队列等待执行。

现在已经定义了两个资源组，可以开始定义选择器规则了。当一个新的查询到达协调器时，它被分配到一个特定的组。下面是一个例子：

```
"selectors": [
    {
      "queryType": "DATA_DEFINITION",
      "group": "ddl"
    },
    {
      "group": "ad-hoc"
    }
  ]
```

第一个选择器匹配类型为 `DATA_DEFINITION` 的所有查询，并将它分配到 `ddl` 资源组。第二个选择器匹配所有的查询，并将它们放到 `ad-hoc` 资源组。

选择器的顺序非常重要，因为它们是依次处理的，直到某一个选择器匹配成功并将查询分配到资源组。若要成功匹配，查询必须匹配所有指定的属性。如果我们调换两个选择器的顺序，则包括 DDL 在内的所有的查询都会被分配给 `ad-hoc` 资源组，而 `ddl` 资源组不会分配到任何查询。

12.8.1　资源组的定义

让我们进一步了解资源组的以下属性。

`name`

资源组名，必填项。选择器规则会使用这个名称引用资源组以分配查询。

`maxQueued`

资源组查询队列的最大长度，必填项。

`hardConcurrencyLimit`

资源组允许的最大并发查询数量，必填项。

`softMemoryLimit`

并发运行的查询可以使用的分布式内存最大值，必填项。一旦达到此限制，在内存释放之前，新的查询都要在队列中等待。属性值可以使用绝对值（GB）和百分比（%）。

`softCpuLimit`

在一个时间段（由 `cpuQuotaPeriod` 属性定义）中可以使用的 CPU 时间的软限制，可选项。一旦达到此限制，正在执行的查询会被施加一个惩罚。

`hardCpuLimit`

在一个时间段（由 `cpuQuotaPeriod` 属性定义）中可以使用的 CPU 时间的硬限制，可选项。一旦达到这个限制，就会拒绝查询直到下一个**配额时间段**（quota period）。

`schedulingPolicy`

在资源组的查询队列中选择一个新的查询来调度和处理的策略，参见 12.8.2 节。

`schedulingWeight`

与 `schedulingPolicy` 一起使用的可选属性。

`jmxExport`

指示资源组是否通过 JMX 暴露信息的标志，默认是 `false`。

`subGroups`

用于其他嵌套资源组的容器。

12.8.2　调度策略

上述列表中 `schedulingPolicy` 属性的值可以配置为在以下几种模式中运行的值。

公平调度

将 `schedulingPolicy` 设为在先进先出（FIFO）队列中的公平调度查询。如果此资源组包含子资源组，就会交替使用队列中有查询的子资源组。

优先级调度

将 `schedulingPolicy` 设置为 `query_priority` 会基于优先队列调度查询。查询的优先级由客户端的 `query_priority` 会话属性指定（参见 8.8 节）。如果资源组有子资源组，则

子资源组也必须指定 `query_priority`。

权重调度

将 `schedulingPolicy` 设置为 `weighted_fair`，这用于选择启动下一个查询的子资源组。此策略需要配合 `schedulingWeight` 属性使用：调度器将基于子资源组的 `schedulingWeight` 按比例选择查询。

12.8.3 选择器规则定义

选择器规则（selector rule）必须定义 `group` 属性，此属性决定将查询分配到哪个资源组。最好将文件中的最后一个选择器设置为只有 `group` 属性的规则，它将为所有查询显式指定一个默认组。

可以使用可选属性和正则表达式定义选择器规则。

`user`

匹配用户名。可以使用正则表达式来匹配多个名称。

`source`

匹配查询来源，例如 `presto-cli` 或 `presto-jdbc`。可以使用正则表达式进行匹配。

`queryType`

匹配查询类型，可选项包括 `DATA_DEFINITION`、`DELETE`、`DESCRIBE`、`EXPLAIN`、`INSERT` 和 `SELECT`。

`clientTags`

匹配提交查询的客户端指定的客户端标签。

你可以使用 `--source` 或 `--client-tags` 选项在 Presto CLI 上指定查询来源和客户端标签：

```
$ presto --user mfuller --source mfuller-cli

$ presto --user mfuller --client-tags adhoc-queries
```

12.9 小结

恭喜！你已经和 Presto 一起走过了很长的路。在本章中，你了解了监控、调优和配置 Presto 的诸多细节。当然，总是有很多知识要学，这些技能通常会随着实践不断提高。确保你和其他用户之间保持联系以交换想法，学习别人的经验以及加入 Presto 社区（参见 1.4.3 节）。

在第 13 章中，你将了解到其他人用 Presto 取得的成绩。

第 13 章

真实世界的案例

如 1.5 节所述，Presto 最初是由 Facebook 开发的。自 2013 年开源以后，很多行业和公司开始使用它。

在本章中，你会看到一些关键数字和特性，它们能让你对 Presto 的潜力有更好的认识。记住，这些公司都是先从小集群开始使用，并在使用的过程中逐渐学习的。当然，很多更小或更大的公司也在使用 Presto。这里给出的数据只是让你对 Presto 可以扩展到什么地步有一个大概的认识。

此外，别忘了很多平台会内嵌 Presto。这些平台可能不会暴露内部使用了 Presto 这件事，也通常不会暴露用户数、架构和其他特性。



本章引用的统计数值都基于公开可获取的信息。本书仓库包含指向数据源的链接，如博客文章、会议上公布的演示文稿和视频等（参见 1.4.6 节）。在你阅读这本书时，这些数据可能已经过时或不再精确。然而，随着更多的人使用 Presto，本章的内容可以帮助你理解 Presto 能做到什么，以及其他用户如何成功地部署、运行和使用它。

13.1 部署和运行时平台

将 Presto 部署在哪里以及如何部署是一个重要的因素，它会影响到从底层的技术需求到面向上层用户的方方面面。从技术的角度来看，影响包括运行 Presto 需要人为参与的程度、使用的工具和运维技术。从用户的角度来看，平台会影响诸如总体性能、演进速度和对不

同需求的适应性等方面。

最后，其他方面也可能影响你的选择，比如使用你所在公司内部的特定平台以及预期成本。下面给出了一些常规做法。

下面是决定在哪里运行 Presto 所需考虑的点。

- 物理机组成的集群越来越少见。
- 虚拟机是目前最常用的平台。
- 容器的使用正逐渐成为标准，很可能取代虚拟机的位置。

作为一个现代的、横向扩展的应用程序，Presto 可以运行在所有的常见部署平台：

- 基于 OpenStack 的私有云；
- 包括 AWS、GCP 和 Azure 在内的公共云供应商；
- 混合部署。

下面是一些案例：

- Lyft 在 AWS 上运行 Presto；
- Pinterest 在 AWS 上运行 Presto；
- Twitter 在私有云和 GCP 上混合部署 Presto。

业界迁移到容器和 Kubernetes 的趋势也影响了 Presto 的部署方式。越来越多的 Presto 集群运行在私有或公有 Kubernetes 集群上。

13.2　集群规模

一些较大的用户运行的 Presto 集群的规模即使在这个到处都是大数据的时代也十分惊人。多年来，Presto 已被证明可以在生产环境下大规模部署。

到目前为止，我们主要谈论的是单个 Presto 集群，但已多次提示过你可以运行多个集群。

在现实世界中大规模使用 Presto 的情况大多数是使用多集群部署。就 Presto 配置、数据源和用户访问等几个方面来说，有很多种使用多集群的方法：

- 相同配置；
- 不同配置；
- 混合配置。

使用配置相同的集群有以下优势。

- 使用负载均衡器可以让你在不影响用户使用的情况下升级集群。

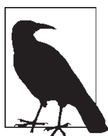
- 协调器故障不会导致系统停止服务。
- 为了提高集群性能所进行的横向扩展可以包含来自不同运行时平台的集群。例如，你可能有一个长期使用的内部 Kubernetes 集群和一个部署在公共云上的临时 Kubernetes 集群。后者可用于在高峰时段分担任务负载。

另外，相互独立的集群可以让你清晰地隔离不同的使用场景并在不同方面调优集群：

- 可用数据源；
- 集群距离数据源的位置；
- 不同的用户和访问权限；
- 调整工作节点和协调器的配置以适应不同的使用场景，例如，针对 ad-hoc 交互式查询和长时间运行的批量任务进行调整。

下列公司运行着多个不同的 Presto 集群：

- Comcast
- Facebook
- Lyft
- Pinterest
- Twitter



本章提到的大多数其他组织很可能也使用了多个集群，但没有将它们列入的原因是，我们尚未找到可以证明这一点的公开引用。

看过集群的数量之后，我们来看一下节点的数量。有一些部署的规模真的很大，另一些部署则是你在未来可能达到的规模。

- Facebook：多个集群共使用了超过 10 000 个节点。
- FINRA：超过 120 个节点。
- LinkedIn：超过 500 个节点。
- Lyft：超过 400 个节点，每个集群 100~150 个节点。
- Netflix：超过 300 个节点。
- Twitter：超过 2000 个节点。
- Wayfair：300 个节点。
- Yahoo! Japan：600 个节点。

13.3　Hadoop/Hive迁移的使用场景

从 Hive 迁移过来可能仍是 Presto 最常见的使用场景，这让我们可以高效地执行 SQL，并与之前保持兼容。而之所以迁移，往往是为了能够查询 HDFS 以外的数据，如 Amazon S3 及其兼容系统，以及很多其他分布式存储系统等。

Presto 的第一个使用场景往往成为在其他方面更广泛使用 Presto 的跳板。

使用 Presto 来查询这些系统中的数据的公司包括 Comcast、Facebook、LinkedIn、Twitter、Netflix 和 Pinterest。下面是一些公司使用 Presto 处理的数据量。

- Facebook：300PB。
- FINRA：超过 4PB。
- Lyft：超过 20PB。
- Netflix：超过 100PB。

13.4　其他数据源

在典型的 Hadoop/Hive/ 分布式存储使用场景之外，Presto 也越来越多用于许多其他数据源。这既包括 Presto 项目官方提供的连接器支持的数据源，也包括很多开源项目、内部开发项目和商业供应商等提供的第三方连接器支持的数据源。

下面是一些例子。

- Comcast：Apache Cassandra、Microsoft SQL Server、MongoDB、Oracle、Teradata
- Facebook：MySQL
- Insight：Elasticsearch、Apache Kafka、Snowflake
- Wayfair：MemSQL、Microsoft SQL Server、Vertica

13.5　用户和流量

最后，我们来看看这些部署 Presto 的用户及其通常的查询数量。用户包括使用仪表盘和 ad-hoc 查询的商业分析师、挖掘日志文件和测试结果的开发者等。

下面是一小部分用户量的统计。

- Arm Treasure Data：大约 3500 名用户。
- Facebook：每天使用的员工超过 1000 名。
- FINRA：超过 200 名用户。
- LinkedIn：大约 1000 名用户。

- Lyft：大约 1500 名用户。
- Pinterest：月活跃用户超过 1000 名。
- Wayfair：超过 200 名用户。

查询的规模从小而简单的查询到大而复杂的分析查询，乃至 ETL 类型的工作负载都有。因此，查询的数量本身只能透露部分信息，但你仍能从中了解 Presto 处理任务的规模。

- Arm Treasure Data：每天大约 600 000 条查询。
- Facebook：每天超过 30 000 条查询。
- LinkedIn：每天超过 20 000 条查询。
- Lyft：每天超过 100 000 条查询，每月超过 1 500 000 条查询。
- Pinterest：每月超过 400 000 条查询。
- Slack：每天超过 20 000 条查询。
- Twitter：每天超过 20 000 条查询。
- Wayfair：每月多达 180 000 条查询。

13.6　小结

Presto 可以支持的运行规模和覆盖的使用场景很广。可以看到，Presto 广泛应用于多个行业。作为新用户，你可以确信 Presto 能扩展到你所需的规模并能处理你的工作负载。

我们鼓励你借助各种资源进一步了解 Presto，特别是加入 Presto 社区和参与社区活动。

第 14 章

总结

无论最初深入学习 Presto 的动机是什么，你现在已经取得了很大的进步。恭喜你！

本书覆盖了 Presto 的很多相关内容，从 Presto 概览和它的使用场景，到安装和运行 Presto 集群。你了解了 Presto 的架构与它如何优化和运行查询。

你学习了多种连接器，可以通过它们将 Presto 和分布式存储系统、RDBMS 和许多其他数据源连接起来。借助 Presto 的标准 SQL 支持，你可以查询它们，甚至还能在一个查询里组合来自不同数据源的数据（即联邦查询）。

之后，你掌握了在生产环境大规模运行 Presto 的后续步骤，以及 Presto 在一些组织当中的实际应用。

希望你已经迫不及待地将这些知识用于实践，我们期待着在社区聊天室收到你的消息（别忘记查看 1.4.3 节）。

感谢阅读，Presto 社区欢迎你加入！

马特、曼弗雷德和马丁

关于作者

马特·富勒（Matt Fuller）是 Presto 公司 Starburst 的联合创始人。在创立 Starburst 之前，他是 Teradata 的工程负责人，为公司构建了全新的 Center for Hadoop 部门，其中主要的工作就是将 Presto 引入企业市场。自 2015 年开始，马特就管理一个向 Presto 开源项目贡献代码的团队，并主导内部的 Presto 产品路线图。Teradata 的这个团队后来发展成了 Starburst 公司。

在加入 Teradata 之前，马特是 Vertica 早期的工程师，他在这里参与构建了 Vertica 的查询优化器。马特也是数据库领域顶级会议 VLDB 论文作者并在数据库管理系统领域拥有多项美国专利。

曼弗雷德·莫泽（Manfred Moser）是开源社区拥护者、作家、培训师，并在 Starburst 担任软件工程师。他长期致力于开发开源软件并促进其社区的发展。曼弗雷德是 Apache Maven 的贡献者，曾写过 *The Hudson Book* 等书，并持续积极参与开源社区和开源项目。他是 CI/CD、云原生、敏捷开发，以及其他软件开发工具和流程方面的资深培训师和技术会议演讲者。曼弗雷德已为 Walmart Labs、Sonatype 和 Telus 等公司累计培训了超过 20 000 名开发者。

曼弗雷德拥有数据库背景，曾在 RDBMS 领域进行数据库和相关应用程序的设计，并作为商业智能顾问写过成千上万行的 SQL。现在，他很高兴可以使用 Presto 并致力于让更多的人知道 Presto 有多么优秀。

马丁·特拉韦尔索（Martin Traverso）是 Presto 软件基金会的联合创始人和 Starburst 的首席技术官。在创立 Starburst 之前，马丁是 Facebook 的软件工程师。在 Facebook 时，他注意到对快速交互式 SQL 分析的需求，并和其他三位工程师一起创造了 Presto，之后带领 Presto 开发团队于 2013 年将其投入生产环境。同年秋天，Presto 开放源代码，从此在 Facebook 公司内外得到广泛采用。

在加入 Facebook 之前，马丁在 Proofpoint 和 Ning 担任架构师，在那里他主导了无数复杂的企业应用程序和社交网络应用程序的开发与架构设计。

关于封面

本书封面上的动物是南方豹蛙（Lithobates sphenocephalus）。这种豹蛙原生于北美洲东部，分布于从纽约南部到佛罗里达州的大片区域，向西可达俄克拉何马州东部。在夏季，很容易在淡水栖息地和潮湿植被附近找到它们。

南方豹蛙以在棕绿色身体上遍布独特的圆点而闻名。它的头很尖，腿很长，从眼睛到臀部有一条由隆起的皮肤形成的明显的脊。雌性南方豹蛙通常比雄性体型大，二者都没有趾垫。

南方豹蛙是夜行动物，通常捕食小型无脊椎昆虫和蜘蛛，但某些体型较大的南方豹蛙偶尔也会吃掉小型脊椎动物。在白天，它们隐蔽在淡水水体附近的植被中。它们经常连续跳跃三次，并可以跳得很高。它们平时单独行动，只有在交配季节才会大量群聚。它们有一对发声袋，膨胀为球形时可以发出短促的喉颤音，非常像小鸡在咕咕叫。南方豹蛙的鸣叫声比其许多近亲传得远。

鸟类、水獭、某些鱼和多种蛇会捕食南方豹蛙。它们还遭到大量抓捕以用作诱饵或用于科研和教学目的。尽管在本书写作时，南方豹蛙的濒危状态是“无危”，但 O’Reilly 图书封面上的很多动物是濒危物种，它们对世界很重要。

封面图片由 Jose Marzan 基于 *Wood’s Natural History* 上的黑白版画绘制。

O'REILLY®

Presto实战

作为高性能分布式SQL查询引擎，Presto能够针对不同的数据源高效、快速地执行交互式分析。大数据的兴起使得数据存储机制多样化。面对标准不一的存储系统，你可以利用Presto轻松打破壁垒，连通数据孤岛。

本书由Presto项目创始成员参与执笔。你将学会用简单的Presto语句快速查询多个数据源，把握SQL-on-Anything的精髓。在任何规模、任何存储系统、任何环境中，你都能发挥SQL的威力。

- **入门：探索Presto的用例，学习安装、配置和使用Presto。**
- **进阶：深入理解Presto的架构，在生产环境中部署Presto，学习连接器实例和SQL高级特性。**
- **实践：了解如何在生产环境中使用Presto，保证安全性，监控负载，进行查询调优，与其他工具集成。**

马特·富勒（Matt Fuller），Starburst公司联合创始人。

曼弗雷德·莫泽（Manfred Moser），开源社区拥护者、技术作家、培训师，Starburst公司软件工程师。

马丁·特拉韦尔索（Martin Traverso），Presto项目创始成员，Presto软件基金会联合创始人，Starburst公司首席技术官。

"你可以通过这本书学习从使用场景到如何大规模运行Presto集群的重要知识。"
——**Ashish Kumar Singh**
Pinterest大数据查询处理平台技术负责人

"如果想构建现代化的分析技术栈，那么这本书值得一读。"
——**Jay Kreps**
Apache Kafka联合创作者

"无论你之前是否使用过Presto，相信你都能从这本书受益。"
——**曹伟**
PolarDB创始人，阿里巴巴研究员

"开源大数据爱好者可以先不要急着掉进源代码细节里，而是从这本书起步，搞懂大数据。"
——**郑锴**
Apache Hadoop PMC成员，
阿里巴巴高级技术专家

"Presto在大数据领域的重要性不言而喻，这本书正是大家期待的那本'官方指南'。书中第三部分为企业级应用做了详细解答，是一大亮点。"
——**腾讯Presto Oteam团队**

PROGRAMMING LANGUAGES / SQL

封面设计：**Karen Montgomery　张健**

图灵社区：**iTuring.cn**

分类建议	计算机/大数据/SQL

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版



扫码领取
随书代码资料



ISBN 978-7-115-56005-6

定价：99.00元